NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

RECOMMENDATIONS FOR SECURE INITIALIZATION ROUTINES IN OPERATING SYSTEMS

by

Catherine Dodge

December 2004

Thesis Co-Advisors: Cynthia E. Irvine
Thuy D. Nguyen

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			*Form Approved OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** December 2004	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Recommendations for Secure Initialization Routines in Operating Systems			**5. FUNDING NUMBERS**
6. AUTHOR(S) Catherine A. Dodge			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

While a necessity of all operating systems, the code that initializes a system can be notoriously difficult to understand. This thesis explores the most common architectures used for bringing an operating system to its initial state, once the operating system gains control from the boot loader. Specifically, the ways in which the OpenBSD and Linux operating systems handle initialization are dissected.

With this understanding, a set of threats relevant to the initialization sequence was developed. A thorough study was also made to determine the degree to which initialization code adheres to widely accepted software engineering principles. Based upon this threat analysis and the observed strengths and weaknesses of existing systems, a set of recommendations for initialization sequence architecture and implementation have been developed. These recommendations can serve as a guide for future operating system development.

14. SUBJECT TERMS Computer Security, Initialization, Bootstrap, Assurance, Modularity, Layering, Coupling, Cohesion, Common Criteria			**15. NUMBER OF PAGES** 133
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

RECOMMENDATIONS FOR SECURE INITIALIZATION ROUTINES IN OPERATING SYSTEMS

Catherine A. Dodge
Civilian, Naval Postgraduate School
B.A., Wellesley College, 2000

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
December 2004

Author:	Catherine Dodge
Approved by:	Dr. Cynthia E. Irvine Thesis Co-Advisor
	Thuy D. Nguyen Thesis Co-Advisor
	Dr. Peter J. Denning Chairman, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

While a necessity of all operating systems, the code that initializes a system can be notoriously difficult to understand. This thesis explores the most common architectures used for bringing an operating system to its initial state, once the operating system gains control from the boot loader. Specifically, the ways in which the OpenBSD and Linux operating systems handle initialization are dissected.

With this understanding, a set of threats relevant to the initialization sequence was developed. A thorough study was also made to determine the degree to which initialization code adheres to widely accepted software engineering principles. Based upon this threat analysis and the observed strengths and weaknesses of existing systems, a set of recommendations for initialization sequence architecture and implementation have been developed. These recommendations can serve as a guide for future operating system development.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank my parents for their love and support. I would also like to thank Paul Bugala for being by my side through this entire journey, even when only in spirit.

I would like to thank my advisors for helping me learn so much, both about the world of computer security and about myself.

And a final thank you to the students of the 500 lab who were always ready with a dose of levity.

This material is based upon work supported by the National Science Foundation under Grant No. DUE-0114018. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

When designing and developing a secure system, often the overall security model depends on proving that the operating system can only transition into another secure state from an initial secure state. This chain of reasoning provides the required level of assurance that the system will behave as expected. Questions arise when one begins to examine how the system is supposed to have arrived at this initial secure state. There are generally trust relationships involved upon which this initial secure state relies. These trust relationships are at times explicit in the system design but often arise out of unstated assumptions about the environment that each piece of code inherits.

This thesis explores the most common architectures used for bringing an operating system to its initial state, once the operating system gains control from the boot loader. While a necessity of all operating systems, the code that initializes a system can be notoriously difficult to understand. This can be attributed to a number of factors including the degree to which assembly language is used and the kind of low-level, hardware-specific operations that are being performed. While such low-level coding creates a more exact mapping between the code and the processor's operations than using a high-level programming language would, it also makes the code more difficult for humans to understand. Another issue is the degree to which these very low-level operations are platform dependant. Due to the processor-specific and device-specific expertise required to understand this specialized code, it is likely the least examined piece of any operating system.

The motivation of this study is to better understand and document the techniques used in the initialization phase of open source operating systems. The common functions that need to be performed during the initialization phase of any operating system are delineated, such as hardware initialization and the creation of kernel structures. This includes all operations that take place from the time that the operating system gains control from the boot loader (e.g. GRUB, LILO) to the point at which the operating system enters its main management loop. The OpenBSD and Linux operating systems have been chosen for analysis.

The focus of this analysis will be to understand the overall initialization architecture and determine the type of security risks that could exist. The risks are determined by examining the threats that the initialization sequence must counter. The initialization code is also analyzed with respect to a number of widely accepted software engineering principles. Based upon this threat analysis and the observed strengths and weaknesses of existing systems, a set of recommendations have been developed which outline the best practices for initialization design and implementation.

These recommendations should benefit designers of both open source and high assurance operating systems. By more widely disseminating information and knowledge about this topic, those designing future high assurance operating systems for the military can readily educate themselves. In addition, application of the proposed recommendations will result in more explicitly defined trust relationships. Overall, the desired effect is systems that are more resilient against malicious attackers. The directions for future research outlined in the final chapter will also serve to spark new thinking about the design and implementation of operating systems.

I. INTRODUCTION

The increasing reliance on technology within the Department of Defense has brought into focus the need to secure these assets. Increasingly it is being recognized that a goal of perfect security is not attainable and rather the paradigm of risk management should be adopted. Central to the notion of risk management is the "defense-in-depth" strategy. The goal of defense-in-depth is to deploy a number of complementary technological and policy measures in order to reduce the risk of decreased confidentiality, integrity or availability of a system.

One essential component of such a strategy is the operating system in use. Extensive work has gone into researching how high assurance systems can and should be developed. While this body of work is quite large, little has been written about the issue of booting and initialization. On the one hand, one could assume that this lack of information indicates that it is an unimportant piece in the overall picture that is not worth exploration. On the other hand, it could be the case that the way in which a system is initialized is vitally important but simply not well understood. The goal of this thesis is to explore these questions and come away with a deeper understanding of the role that the initialization sequence plays in determining the degree of assurance provided by an operating system.

A. SCOPE AND METHODOLOGY

In order to research and document the ways in which initialization routines have been implemented, two open source operating systems were chosen for study, OpenBSD and Linux. A more in-depth study of OpenBSD was performed, while the way in which Linux does initialization was also explored in order to provide contrast. The study of a high assurance or closed-source commercial operating system would have also been useful. But the proprietary nature of most commercial systems would have limited the degree to which any results could have been published. There are currently no open implementations of an operating system that adheres to high assurance principles that could have been used.

When exploring the two operating systems chosen, the focus of study was the overall design and sequencing of the initialization routine. While the source code served as the primary focus of study, the intent was not to complete a thorough source code audit seeking actual vulnerabilities. To do so would have required a much more in-depth understanding of the entire implementation of both operating systems than would have been possible in the given timeframe. Rather the focus was on overall design characteristics. In keeping with this methodology, comments in the code were for the most part taken at face value.

B. DEFINITIONS

In order to clarify the terminology used throughout this document, a number of definitions are provided below.

daemon – A program that is typically started when the operating system is booted and continues to run, waiting for a specified action to occur, throughout the life of the system.

gate – In order to provide access between code segments of differing privilege levels, the Intel x86 architecture defines a special type of descriptor, called a gate descriptor or simply a gate. There are four different types of gate descriptors: call gates, trap gates, interrupt gates and task gates. Call gates are used to manage the transfer of program control between privilege levels when it is not exception or interrupt related. A call gate may exist in either the GDT or in an LDT. Trap and interrupts gates are similar to call gates, but are used specifically for invoking exception and interrupt handlers, and exist in the IDT. Task gates are used for switching between tasks, defined by different task-state segments. A task gate may exist in the GDT, an LDT or the IDT.

global descriptor table (GDT) – Used to arbitrate all accesses to memory segments while the Intel x86 chip operates in protected mode. This table consists of a list of segment descriptors, each of which defines the base address of a segment, access rights, type and usage information. A segment selector is associated with each descriptor. In order to access global memory, i.e. memory that is available to all tasks, a segment selector and offset must be specified. These are used to index into the GDT to provide the segment descriptor which, in turn, identifies the required segment. The linear address of the base of the GDT is contained in the GDT register (GDTR).

initialization sequence – The steps that are executed by the operating system from the point at which control is handed over by the boot loader to the point at which the first process (Process 0 on UNIX systems) finishes initialization and begins executing in a loop its primary function. Consideration of user space initialization, typically handled by the *init* program on UNIX systems, is excluded. The terms *initialization routine* and *initialization code* are used interchangeably with this term.

interrupt descriptor table (IDT) – For an Intel x86 system, a single IDT must be defined that determines how interrupts and exceptions are handled. The table is made up of task gate, interrupt gate and trap gate descriptors. Interrupt and exception values provide an index value into the IDT, locating a gate descriptor which identifies the location of the code segment containing the interrupt procedure that should be run.

kernel – Refers to the core functions of an operating system, which may be separated from the functions of the operating system deemed auxiliary and run as its own program. In the context of UNIX systems the term kernel is equivalent to the term operating system, as a monolithic design is used. This term is often used to distinguish the operating system from supporting software, e.g. "The Linux kernel does not specify which daemons must be run on a system, that is distribution dependent."

kernel space – Code running in kernel mode, which on an x86 system means running within privilege level 0, is said to be running inside kernel space. Kernel mode is also referred to as "supervisor mode."

kernel thread – A process that shares kernel data structures with Process 0, the first process created on the system. This term is used as little as possible in this thesis in favor of more precise terms, e.g. Process 1.

local descriptor table (LDT) – Each task, defined by a TSS, on a system has an associated LDT. This table consists of a list of segment descriptors, each of which defines the base address of a segment, access rights, type and usage information. A segment selector is associated with each descriptor. In order to access processor-specific memory a segment selector and offset must be specified, which index into the LDT to provide the segment descriptor for the required segment. The GDT must contain a separate segment selector and segment descriptor table entry for each LDT segment defined on the system. The linear address of the LDT is contained in the LDT register (LDTR).

operating system – The software program that virtualizes access to the underlying hardware of a system in order to provide a standard exported interface for user programs to utilize the hardware in a shared manner. Oftentimes the operating system is divided into separate software components, a kernel which handles the core operations and other programs to handle items such as file system management. This distinction is not made in this thesis as the operating systems explored here have a monolithic architecture with all operating system activities handled within a single program.

privilege level – Indicates the privilege level of the currently executing program or procedure. For the Intel x86 chip, it is determined by bits 0 and 1 of the CS segment register. The privilege level can have a value of 0 to 3. Most sensitive instructions on the Intel architecture, such as loading the global descriptor table (*lgdt*), can only be executed by programs with a privilege level of 0, often referred to as supervisor or kernel mode.

process – Refers to the UNIX notion of a process. Each process may or may not have its own TSS, defining it as a *task* in the Intel vocabulary, depending on the implementation. For example, the Linux 2.4 kernel only defines a single TSS for a single processor system, regardless of how many processes are created on the system. OpenBSD creates a TSS for each process that is created on a system.

protected memory mode – This operating mode, introduced with the Intel 286 processor, provides features to enhance multitasking and support memory protection, paging and virtual memory. To access the memory segments defined in protected mode, a segment selector stored in one of the segment registers (CS, SS, DS, ES, FS and GS) serves as an index into a descriptor table. The descriptor that the selector points to provides the physical location information for that segment, in addition to access control information.

real memory mode – This operating mode provides a programming environment like that of the Intel 8086 processor, with the ability to switch to protected mode. In real mode the selector values stored in the segment registers CS, SS, DS, ES, FS and GS serve as the upper 16-bits of the 20-bit linear address needed to access memory.

segment – The Intel 386 processor, as well as newer Intel processors, has the ability to divide the processor's addressable memory space into protected address pieces called segments. Each program on a system can be assigned a set of segments and the processor will enforce the boundaries between the segments belonging to different processes.

segment selector – A segment selector is used to enforce segmentation on Intel x86 processors. "A segment selector is a 16-bit identifier for a segment. It does not point directly to the segment, but instead points to the segment descriptor that defines the segment." [INT03-3] A segment selector consists of the following data items: an index value which determines which of the 8192 descriptors in the GDT or LDT to select, a table indicator flag that indicates whether the desired descriptor is in the GDT or the current LDT, and a requested privilege level (RPL) value, 0 thru 3, used to ensure that more privileged code does not access a segment on behalf of an application unless the application itself has access to the segment.

segment descriptor – A segment descriptor is used to enforce segmentation on Intel x86 processors. A segment descriptor defines a given segment in memory. Found in the GDT or LDT, a descriptor "specifies the size of the segment, the access rights and privilege level for the segment, the segment type, and the location of the first byte of the segment in the linear address space (called the base address of the segment)." [INT03-3]

task – This term is used by Intel to define the unit of program execution. "All program execution in protected mode happens within the context of a task, called the current task." [INT03-3] Once in protected memory mode, a task is always running. A single task is defined by an associated TSS. The term "task" is not used in this thesis with this implied technical definition. Instead the term "process" is favored as this is more descriptive since exclusively UNIX-like systems are discussed here and each process does not necessarily have a corresponding TSS.

task-state segment (TSS) – "defines the state of the execution environment for a task. It includes the state of the general-purpose registers, the segment registers, the

EFLAGS register, the EIP register, and segment selectors and stack pointers for three stack segments (one stack each for privilege levels 0, 1, and 2). It also includes the segment selector for the LDT associated with the task and the page-table base address." [INT03-3]

user space – Code running in user mode, which on an x86 system means code running within privilege level 3, is said to be running inside user space.

C. THESIS ORGANIZATION

This thesis is organized into seven chapters beyond this introductory one, Chapter I. Chapter II describes in a generic manner the way in which a typical x86-based PC boots, from power-on to OS initialization. Chapter III describes how the OpenBSD operating system is booted and initialized. Chapter IV describes the same sequence of operations within the Linux operating system. Chapter V explores the types of threats that are relevant to the initialization sequence of an operating system, using OpenBSD as a point of reference. Chapter VI outlines the degree to which OpenBSD adheres to generally accepted security engineering practices. The culmination of this research is Chapter VII, which describes a number of proposed recommendations for building a robust initialization routine, based on the threats and security engineering principles identified as relevant in the preceding chapters. Chapter VIII focuses on summarizing the results and outlining directions for future work.

II. OVERVIEW OF THE BOOT PROCESS

A. INTRODUCTION

The intent of this chapter is to give the reader a general overview of the boot process of an Intel x86 based personal computer. This is required to understand how the initialization routine provided by the operating system fits into a larger picture. Generally speaking, there are three stages to the bootstrapping process. When a PC is initially powered on it is the BIOS (Basic Input-Output System) that runs first, followed by a boot loader and finally the operating system initialization routine. The logical execution sequence of these components is shown in Figure 1 below. The physical location of each of these boot-related components is shown in Figure 2. Each of these three parts will be discussed in this chapter, as well as how they interface with one another to bootstrap the operating system.

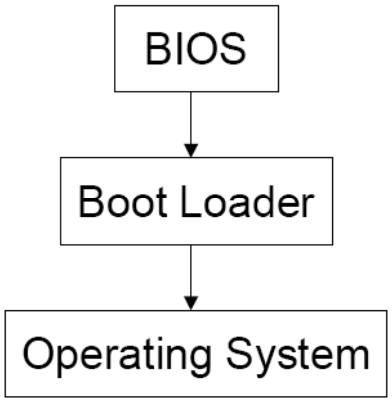


Figure 1. Logical Execution Sequence from Boot to Runtime on an Intel-based Personal Computer

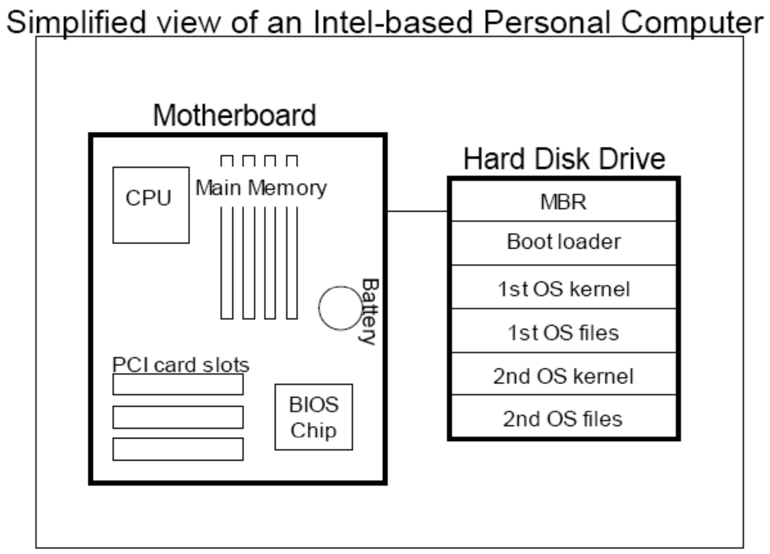


Figure 2. Physical Location of Initialization-related Components on an Intel-based Personal Computer

B. THE BIOS

The BIOS is the first code to be executed by the processor upon boot. When power is initially applied to the computer this triggers the RESET pin on the processor. This in turn causes the processor to read from memory location 0xFFFFFFF0 and begin executing the code found there. This address is mapped to the Read-Only Memory (ROM) containing the BIOS. The BIOS is responsible for polling the hardware and setting up an environment that is capable of booting the operating system. The BIOS functionality can be broken into three areas: Power On Self Test (POST), Setup and Boot.

The main function of the POST is to test and initialize the various hardware components detected by the BIOS. In the case of PCI devices, the POST routine is where devices are assigned specific Interrupt Request Line (IRQ) values. Once the BIOS begins running, it scans the system for ROM chips associated with hardware devices, commonly referred to as “option ROMs” [CPQ96]. These option ROMs contain the routines needed to initialize the hardware device. Newer hardware devices have option ROMs that are compliant with the Plug and Play (PnP) standard, as detailed in [CPQ94]. This standard allows for a PnP operating system to perform resource allocation dynamically at runtime. With an older BIOS, the operating system is forced to consider the configuration of devices to be static, having already been determined by the BIOS. With a PnP BIOS, a PnP operating system may directly manipulate the configuration of devices at runtime. Operating systems such as Windows 2000 take advantage of this capability while others, e.g., Linux, do not. Regardless of whether the option ROM is PnP compliant or not, the BIOS makes a far call to offset +03h in each option ROM in order to initialize the device [CPQ96]. The order in which option ROMs are called is also prescribed for a PnP compliant BIOS. A PnP BIOS first initializes the option ROM of the video card, then any option ROMs containing a valid PnP Expansion Header. Finally those option ROMs with legacy formatting are initialized.

As part of the POST routine, hardware devices also register themselves with the BIOS as associated with or "hooked" into certain interrupt values. The routines for doing so are also stored on the option ROMs. For instance, by convention on the Intel platform, all hard drives register with the 13h interrupt. Programs can then invoke this software interrupt to communicate with disk drives, utilizing device driver code within the BIOS. In order to distinguish between the multiple hard drives that may exist in a single computer system, each disk is assigned an identifier number. This identifier becomes important when the BIOS tries to boot the operating system from one of the devices discovered during the POST phase. Depending on the identifier assignment, some drives may be bootable while others may not be. It is important to note here that while older operating systems such as DOS rely heavily on the BIOS to provide an interface to the hardware, newer operating systems are moving away from using BIOS routines to communicate with hardware. Modern operating systems such as Windows 2000 and Linux bypass the BIOS when communicating with hardware devices and instead supply their own 32-bit device drivers once fully running, but do make use of BIOS routines in order to initially load the operating system from disk.

Once these POST operations are complete, the BIOS offers the end user the option of entering into "Setup" mode. It is here that the user may change the configuration of the BIOS, including the boot sequence. Any configuration changes are saved back to the CMOS memory on the motherboard which holds all BIOS settings. This memory draws power from the on-board battery to ensure the persistence of this information.

The last thing the BIOS does is execute the 19h interrupt, which loads the boot sector of the first device that is to be booted from. This is where the boot loader is located, so once the BIOS executes the 19h interrupt it can be said that control has transferred to the boot loader. If multiple hard drives are registered with the BIOS, the hard drive chosen as the boot device is the one that was assigned the identifier 80h. This means that only one hard drive is allowed to be bootable. Other devices, such as CD-ROM drives, can also register themselves as bootable and be put on a list of bootable devices, which is then prioritized by the user during the “Setup” mode.

C. THE BOOT LOADER

Once the BIOS loads the first sector of the boot device into RAM, it begins executing. In the case of a hard drive, this first sector is referred to as the Master Boot Record (MBR). The MBR contains the partition table describing the partitions defined on the hard drive. It also contains a program which will load the first sector of the partition marked as active into RAM and execute it. The size of the MBR is limited to one sector on disk or 512 bytes, since it is located within the first sector of the drive at cylinder 0, head 0, sector 1. This size limitation greatly limits its functionality as well. Typically boot loaders have been highly integrated with the operating system that they support. This integration cuts down on the operations a boot loader must perform, making a 512 byte boot loader feasible. More functionality can be provided by multi-stage boot loaders.

A multi-stage boot loader provides more function and flexibility by working around the 512 byte size limitation. Rather than consisting of a single program which loads the operating system directly, multi-stage boot loaders divide their functionality into a number of smaller programs that each successively load one another. This architecture allows a fairly primitive boot loader, located in the MBR, to load and execute the next stage of the boot loader, a larger and more sophisticated boot loader. Subsequent stages can be located elsewhere on the hard drive and thus are not subject to the single sector size limit. This chaining of boot loaders allows the boot loader functionality to become arbitrarily complex. For example, LILO [LILO] and GRUB [GRUB] are two well-known multi-stage boot loaders. Both are capable of booting a number of operating systems including Linux, Windows and FreeBSD, to name a few. Both LILO and GRUB also have multiple stages, which allow them to first “bootstrap” themselves into operation, then present the user with a number of OS boot options. The user can choose which OS to boot and the boot loader has the functionality required to do so. This provides the user with the ability to install multiple operating systems on a single computer. The user can choose which operating system image to boot after the boot loader has run. This has been the main benefit provided by third party boot loaders.

Boot loader developers have access to the OS source code for any open source operating system, which allows them to understand and emulate how any of these operating systems boot. The fact that the functionality required to boot any open source OS is documented in the source code has enabled LILO and GRUB to support the extensive number of open source operating systems that they do. For proprietary operating systems however, access to the source code is not possible. Reverse engineering a proprietary boot loader would be a possibility, but is likely quite complex. Instead boot loader developers have come up with an alternate solution. The third party boot loader, such as GRUB, can be instructed to invoke the boot loader of the proprietary operating system, which then loads the proprietary OS. This type of “chain loading,” where one boot loader loads and executes another boot loader, leads to a quite clean and effective implementation. Rather than, possibly incorrectly, implementing a reverse engineered version of a proprietary boot loader, chain loading invokes the original boot loader that has been tailored to the proprietary OS. In essence, this approach reuses the code that already exists for booting the OS, rather than building new code from scratch. In this way a user could install an open source operating system alongside a proprietary operating system and use a third party boot loader to allow them to choose which to boot at power on. Such code reuse is a hallmark of good software engineering practices.

While boot loaders can exist as standalone software, as GRUB demonstrates, they are most often tailored to and integrated into a specific operating system. This has led to a situation where each operating system requires its boot loader to behave in a slightly different manner. One movement to define a clear-cut API between the boot loader and operating system is the Multiboot Specification, developed and maintained by the Free Software Foundation [OKU02]. The Multiboot standard defines three main aspects of the interaction between the boot loader and the operating system:

1. The format of the operating system image, as perceived by the boot loader

2. The state of the system when the boot loader starts executing the operating system

3. The format of the information passed by the boot loader to the operating system

The Multiboot specification is quite detailed as to the kind of header that must be included in the operating system image, the values that must be in certain registers (not all registers are specified), and the format of the information passed to the OS. Currently the only programs that fully support the specification are GRUB, the Mach kernel [MACH] and the Fiasco [FIA] operating system, while others programs like Linux are somewhat Multiboot compliant. Sun may be adding Multiboot support for Solaris x86 in the near future, with the intent of using GRUB as the boot loader [GRDV04]. The advantages of such a specification are several. It allows the developers of new operating systems to make use of existing boot loaders without having to write one themselves. Having such a clearly defined interface also adds a level of modularity and transparency to the boot process, in line with widely accepted good software engineering practices. It is hoped that as additional projects use this standard it will become more widely supported, disseminated and understood.

For high assurance operating systems, the additional burden of verifying the integrity of the operating system to be loaded is placed on the boot loader and BIOS. For example to ensure that the boot program has not been subverted, the BIOS could verify its authenticity. In turn, to ensure that the kernel has not been changed, the boot loader would do the verification. A very detailed architecture for providing exactly these types of integrity checks from the BIOS level all the way up through to the operating system was proposed by Arbaugh [ARB99]. Traditionally computer systems treat the correctness and integrity of lower virtual machine layers as axiomatic. The AEGIS boot process proposed by Arbaugh guarantees, rather than treats as axiomatic, the integrity of each layer by ensuring that "(1) the integrity of the lower layers is checked, and (2) transitions to higher layers occur only after integrity checks on them are complete." The AEGIS architecture defines a number of levels that make up the IBM PC bootstrap process. Transitions to higher levels only take place after the preceding level has verified the next level via a cryptographic signature. The lowest level, level 0, consists of “trusted” code, the only code involved in the bootstrap architecture that is not verified at boot time. The AEGIS ROM, which contains digital signatures, public key certificates, and recovery code, resides at this lowest layer as does the minimal amount of BIOS code needed to support integrity verification and recovery. The rest of the BIOS code is located in level 1. At level 2 the expansion ROMs are verified and executed, then return control to the BIOS at level 1. From there the BIOS will verify and then load the boot block found on disk, level 3. The boot block in turn verifies and loads the operating system, level 4, which may then similarly verify any application level programs, level 5. If verification of any of these levels fails, a recovery mechanism is provided for restoring the corrupted level from either the AEGIS ROM or from a trusted network host. Figure 3 below illustrates the organization of these levels.

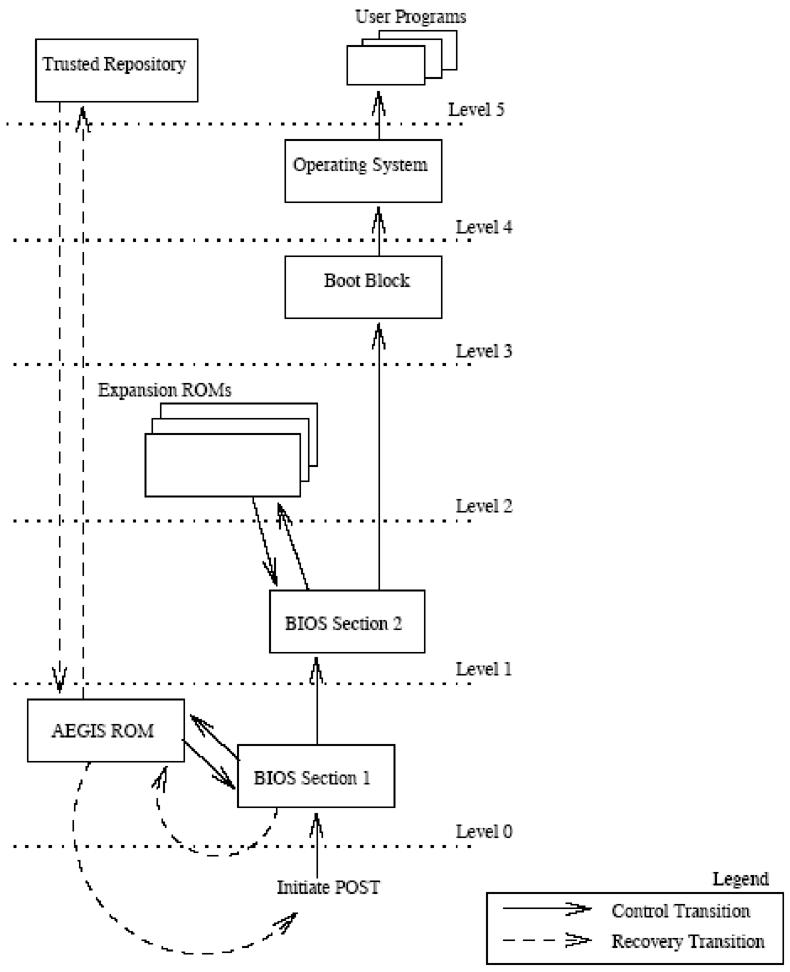


Figure 3. AEGIS Boot Control Flow (From: [ARB99])

While such an architecture can prove the integrity of each level, it cannot prove the correct function of each level, as the author notes. A distinction is made between “trusted” and “trustworthy.” While a piece of code that has passed an integrity check is considered “trusted,” any preexisting flaws that exist in such signed code will still exist after an integrity check. The term “trustworthy” implies that a component has been properly vetted to ensure it is not buggy, either via formal methods or flaw detection techniques. Ensuring that only trustworthy code is signed before being loaded on a system is an equally important component of building a high assurance operating system that is not addressed by the AEGIS architecture but could be a focus of future work.

D. THE OS INITIALIZATION

Once the boot loader has loaded the OS image into memory, control is transferred to the OS and it begins executing. The operating system will go through a series of steps to set up the operating environment that is necessary for the operating system to achieve a coherent initial run state. This "initialization" sequence will not execute again, once the operating system has entered its main scheduling loop.

There are a number of tasks that must be done during this period that are common across operating systems. Data structures needed for kernel operation must be defined before they can be used. Examples are the run queues used to manage scheduling and the structures used to represent files, *vnodes* in the case of OpenBSD. Yet many of the operations that need to be done at this point have interlocking dependencies with one another. While some device drivers may need to create kernel space processes as part of their initialization, support for forking new processes may not yet be available. Thus the initialization code requires an acute awareness of interrelated dependencies.

A large piece of initialization for any operating system is the establishment of virtual memory management. On an Intel-based system this typically involves setting up the global descriptor table (GDT), creating a local descriptor table (LDT), switching the processor into protected memory mode, setting up page directories and enabling paging.

Additional tasks that must be taken care of include device driver initialization and ensuring that interrupts are properly assigned in the interrupt descriptor table (IDT). Another major task that any operating system will handle during initialization is establishing support for various file system types and mounting a root file system. The ultimate goal of many of these initialization tasks is the establishment of support for multi-tasking, the ability to switch between processes, the central purpose of an operating system.

The notion of a *process* is the basic unit of measuring execution in the various versions of UNIX. The definition section clarifies what is meant by the term process in this thesis. One of the conundrums of the bootstrapping sequence is that the code that first executes in the kernel must be able to turn itself into a process that can be swapped in and out of a running state. This is how the appearance of multi-tasking is achieved. Every other process is created during runtime by using kernel functions designed to support process creation, e.g. some kind of *fork()* call. Yet the initial process must explicitly do for itself all the tasks taken care of by a call to *fork()*. This initial process, numbered 0 on UNIX systems, will be referred to as Process 0 in this discussion. Process 0 must be able to self-generate its own process context. Once this context has been established, the system has the capability to suspend and resume execution of Process 0 just as it would any other process. Once established, the role of Process 0 differs by operating system. For example, in OpenBSD Process 0 enters a loop looking for processes that are runnable but not currently resident in memory and swaps them into memory. On the other hand, the Process 0 found on Linux systems executes an idle loop. These behaviors will be discussed more fully in the sections that follow.

While Process 0 is responsible for the bulk of kernel data structure generation on UNIX systems, it is Process 1 that is most familiar to users. Process 1, commonly referred to as the "init" process, is the first process forked from Process 0 and is responsible for the initialization that is done within user space. To avoid confusion, the term "init process" will be avoided in favor of the term Process 1. Depending on the operating system, Process 1 may perform a few tasks before doing its most important job – executing the *init* binary that will perform user space tasks such as starting application daemons and setting network configurations. While, for OpenBSD, Process 0 is the parent of all kernel space processes including Process 1, it is Process 1 that is the parent of all user space processes. The focus of this analysis will be limited to kernel space initialization. Operations performed by Process 1 in user space will not be considered within the context of this thesis. A thorough analysis of user space tasks, and the extent to which they should exist there instead of within kernel space, is material for potential future work.

Once Process 1 has been forked from Process 0, often a number of additional kernel space processes are created to handle additional kernel space tasks. Once all of these are running, the kernel space operating system initialization is complete.

E. SUMMARY

For an operating system to move from the initial power-on state to fully up and running a number of tasks must be handled. As outlined above, these tasks are traditionally divided into three stages on an Intel x86-based PC. The BIOS phase is quite hardware specific and is responsible for only the most low-level tasks. A boot loader, installed on the hard drive, then takes over control and can provide the option to boot various operating systems present on the hard drive. Once an operating system is chosen, the boot loader locates the respective kernel on the hard drive and executes it. The kernel then begins initialization of the data structures it will need to support its operation. Depending on the system, one or more processes may be defined during initialization. Once all have entered their main execution loops, the operating system could be said to be finally running.

In the chapters that follow, the details of how these stages have been implemented will be examined in two different operating systems: OpenBSD and Linux. While both follow the overall design outlined above, noticeable differences also exist. These implementation-specific details will be highlighted. The details of implementation will also inform the latter discussion regarding initialization best practices.

THIS PAGE INTENTIONALLY LEFT BLANK

III. OPENBSD INITIALIZATION

A. INTRODUCTION

While very similar to other versions of UNIX, there are aspects of OpenBSD that make it quite unique. In particular the OpenBSD project has focused a great deal of effort on security. This has largely taken the form of ongoing source code auditing [OBSD04]. In the following discussion, initialization of the system will be traced from the boot loader through the kernel code responsible for bringing up the system to the point where the scheduling loop is entered. As mentioned previously, the operations done by Process 1, the *init* process, fall outside the scope of this analysis.

One feature of OpenBSD that makes it unique is the degree to which the kernel is integrated with the boot loader provided with the OpenBSD distribution. For this reason, it is not currently possible to directly boot OpenBSD using a non-proprietary boot loader like GRUB. Instead OpenBSD must be chainloaded, ensuring the boot loader provided with the distribution is invoked to load the kernel image. Part of the reason that the provided boot loader is so integral is that the second stage boot loader switches the CPU into protected mode. Thus the kernel image expects the CPU to already be in protected mode when it is handed control.

First an outline of the boot process is necessary, which will be followed by a more detailed explanation of the initialization tasks performed by the kernel. Figure 4 provides an overview of the logical execution sequence during the OpenBSD boot process. Release 3.4 of the OpenBSD operating system was used for the analysis presented here and the path of all files referenced is rooted at the */src/sys/* directory within the OpenBSD source tree.

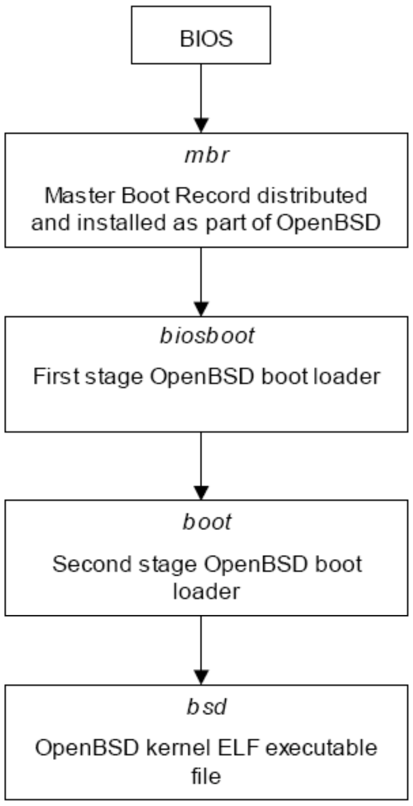

BIOS
mbr
Master Boot Record distributed and installed as part of OpenBSD
biosboot
First stage OpenBSD boot loader
boot
Second stage OpenBSD boot loader
bsd
OpenBSD kernel ELF executable file

Figure 4. OpenBSD Logical Execution Sequence from Boot to Runtime

B. BOOT LOADER INITIALIZATION

The OpenBSD operating system employs a two-stage boot loading process, if one does not count the MBR as a "stage" unto itself. The first stage is handled by a boot loader program called *biosboot*, while the secondary boot loader is called simply *boot*. An MBR image is also included with the OpenBSD image. In the past the *biosboot* program had been used as the MBR itself, but the *biosboot* man page no longer recommends this due to strange interactions based on BIOS peculiarities. At boot time, control moves from the BIOS to the MBR then to the initial boot loader, *biosboot*. This program is installed using a special tool called *installboot*, which patches *biosboot* with information about the location of the second stage boot loader, *boot*. Thus if the location of *boot* were changed, *biosboot* would no longer be able to find the secondary boot loader and the boot process would fail. Once *biosboot* has successfully located and loaded the *boot* program from disk into memory, it will transfer control to this newly loaded program.

The *boot* program handles setting up an environment suitable for transferring control to the kernel image. It also provides an interactive prompt to allow the user to pass additional boot parameters. The main tasks that the *boot* program handles include the following:

1. Switching the CPU into protected mode

2. Probing for console devices and displaying subsequent messages to the discovered consoles

3. Detecting memory, both that reported by the BIOS and extended memory

4. Detecting if the BIOS supports Advanced Power Management (APM)

The program then reads from */etc/boot.conf*, located on the host system, any specified boot parameters that will be used to locate and boot the kernel. Next the user is presented with a prompt which will accept boot parameters interactively. If none are entered, after the timeout period the kernel at the default location */bsd* will be loaded using the parameters found in */etc/boot.conf*. Once the kernel has been loaded, *boot* is finished with its role in the initialization process and control is transferred to the kernel executable file, *bsd*.

C. KERNEL INITIALIZATION

1. Assembly Level Initialization

Upon a successful load of the kernel, execution begins at the *start* label found in the assembly code file */arch/i386/i386/locore.S*. Figure 5 below details the flow of control within the OpenBSD kernel during the initialization phase. The code in this file first loads the parameters passed to it by the boot program via the stack. Thus while there is interaction between the boot program and the kernel, it is handled in much the same way a function call is handled. Parameters that must be passed between them are put on the stack by *boot.* It is assumed that the kernel knows to locate these parameters on the stack, instead of in registers or elsewhere in memory.

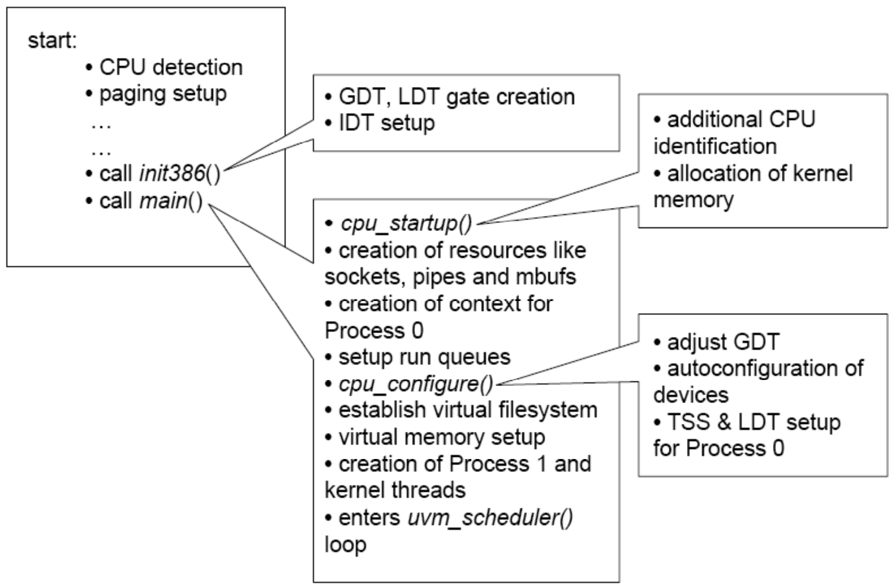

start:
• CPU detection
• paging setup
…
…
• call *init386*()
• call *main*()
• GDT, LDT gate creation
• IDT setup
• additional CPU identification
• allocation of kernel memory
• *cpu_startup()*
• creation of resources like sockets, pipes and mbufs
• creation of context for Process 0
• setup run queues
• *cpu_configure()*
• establish virtual filesystem
• virtual memory setup
• creation of Process 1 and kernel threads
• enters *uvm_scheduler()* loop
• adjust GDT
• autoconfiguration of devices
• TSS & LDT setup for Process 0

Figure 5. Execution Path within *bsd*, the OpenBSD Kernel Executable, During Initialization

The next part of the code determines the exact type of x86 compatible processor the kernel is running on, using various techniques. This information is needed in order to handle processors that do not implement some instructions in hardware and thus require software emulation. A page table directory is subsequently constructed and paging is enabled. After this is finished, the C function *init386()* is called.

The function *init386()* is responsible for handling many of the x86-specific structures that need to be initialized, particularly those related to memory management. This includes creating the GDT, LDT, GDT memory segments and LDT memory segments, as well as initializing the IDT used by the system. Details on the roles of the various x86-specific items named above can be found in the definitions section. The function *pmap_bootstrap()* is then called, which does a first pass of initializing the physical mapping (pmap) module of the memory management. The function *pmap_init()*

finishes initializing the pmap module later, after control enters the *main()* routine[1]. Finished with these tasks, *init386()* returns control to the assembly file *locore.S*. At this point a call is made to *main()*, the machine-independent C language function defined in *init_main.c*.

2. High-level Initialization

The *main()* function calls all of the various subsystem-specific initialization routines needed to prepare the operating system to run. The function itself is completely architecture independent, thus *main()* calls only high-level initialization functions. It is only in the implementation of those functions called within *main()* that machine dependent features are encountered.

Initialization of the current process pointer is the first step. The kernel process that currently has control, is defined to be Process 0. The second step is to initialize the console so that any debugging messages can immediately be written to the screen. Next kernel data structures used for autoconfiguration, virtual memory, disk management and tty management are initialized. The next function called, *cpu_startup()*, handles most of the architecture-specific initialization.

Within *cpu_startup()* the buffer used for writing error messages is first initialized, using memory that was reserved when *pmap_bootstrap()* ran. Next the CPU is identified, which is needed at this point in order to be able to handle processor-specific bugs, such as the "f00f" bug found in older Intel processors [INT97]. The *allocsys()* function that follows is responsible for allocating memory for kernel data structures. Memory is allocated for the entire kernel, to *kernel_map*, and subsequently assigned to specific submaps that were initially defined when *uvm_init()* was called earlier from within *main()* to initialize the structures for virtual memory support. Submaps are subsections of kernel memory that are defined in order to isolate the memory usage of various subsystems from one another. Submaps also break the locking of kernel memory into

[1] The pmap module is responsible for providing translation of high-level memory management functions to the appropriate machine dependent functions. Thus the pmap module implements, for example, architecture-specific values for the three permission levels—read, write and execute—defined by the machine-independent memory protection. Once this basic level of memory support has been established, yet the memory management unit (MMU) has not been fully initialized, it is possible for memory to be dynamically allocated via the *pmap_bootstrap_alloc()* call. Memory allocated in this way will not be managed by the MMU but instead can be considered a hard-wired mapping. Once *pmap_init()* is called and the MMU has been fully initialized, memory can be allocated normally using MMU support.

smaller pieces, since the memory in a submap is managed via the submap it is assigned to, not via the larger *kernel_map*. Thus locking the entire *kernel_map* is not required when only accessing memory within a submap. Once memory is assigned to the various maps, *cpu_startup()* returns control to *main()*.

The next step in *main()* is to call the function *mbinit()* which initializes a pool of *mbuf* data structures. The *mbuf* or memory buffer structure is the foundation of the memory management used to handle networking. The function *soinit()* is subsequently called, which reserves memory space for use by sockets by creating a resource pool named *socket_pool*. The next few high level functions initialize the data structures and resource pools to support timeouts, sysctls, process management and file descriptors. The call to *pipe_init()* similarly sets up a resource pool to support pipe structures.

The next step is to insert data into various lists in order to correctly establish Process 0. While many resources and structures needed to support Process 0 were set up in the earlier assembler code, such as the run-time stack, at that time structures such as the process queues did not exist. Thus it is only at this later point that, for example, an entry for Process 0 can be added to the process ID hash table. Other items that are defined in order to create the context for Process 0 include the *p_stat* and *p_nice* values, which are respectively set to *SRUN*, e.g. currently runnable, and *NZERO*, the default value of 20. Finally Process 0 is given the name of "swapper."

Much of the additional context that needs to be defined for Process 0 involves setting variables within the *proc* structure for Process 0. All of the variables named in this paragraph are defined within the *proc* structure and initialized for Process 0 in the following order:

1. A timeout, named *p_sleep_to*, is established that will timeout when this process returns from calls to any of the *sleep()* family of functions which provide for voluntary context switching.
2. A second alarm timeout, named *p_realit_to*, is created which allows processes to base actions on a "real" timer that decrements in real time.

3. The credentials structure of Process 0, the *pcred* structure *p_cred,* is initialized next. This is used to store the real and effective user id and group id values used by UNIX systems to determine permissions.

4. The resource pool used for allocating all signal structures is initialized via the function call *siginit()*.

5. A new signal structure is assigned to Process 0 and the signals to be ignored are established by setting the *p_sigignore* value, a *sigset_t* structure.

6. A file descriptor table for Process 0 is built by calling *fdinit()* and assigning the returned file descriptor table to the *p_fd* element of Process 0.

7. The resource limits on Process 0 are established. There are nine different resources that can be limited on a per process basis via the *p_limit* value, a plimit structure, such as number of processes and number of open files. Each resource has both a "soft" limit and a "hard" limit defined. The limits on Process 0 are all initially set to RLIM_INFINITY, after which a select few are set to lower limits. For example the soft limit on the number of simultaneous processes is lowered to 80 and the number of open files to 64.

The next section of code within *main()* defines the address space map for Process 0. First *uvmspace_init()* is called which initializes the *vmspace* structure *vmspace0*. The address space for Process 0 is then defined to be this *vmspace0* structure. Lastly, two additional values within the *proc* structure are assigned, p_addr and p_stats. At this point enough of the supporting data structures are in place to be able to assign this process to root, i.e. the number of processes listed as running under user root is incremented to one.

Next the run queues, 32 total, used by the scheduler are initialized and zeroed via a call to *rqinit()*. Then we reach a call to a fairly complex function, *cpu_configure()*. Some of the most important tasks handled by *cpu_configure()* include:

1. Adjusting the GDT to an appropriate size via *gdt_init()*

2. Autoconfiguration of all devices via a call to *config_rootfound()*

3. Initialization of the task-state segment (TSS) and local descriptor table (LDT) for Process 0 by calling *i386_proc0_tss_ldt_init()*

After *cpu_configure()* returns to *main()*, the next function call, *uvm_init_limits()*, establishes limits for Process 0 on virtual memory usage. This function is only called once in the kernel code, as every other process will inherit these limits from Process 0. Next support for NFS (Network File System) is enabled, if the respective kernel configuration option was defined. Support for the virtual file system is then established via a call to *vfsinit()*. This function sets up the *vnode* management structures and prepares the system for being able to handle any type of file system. The *vnode* structure is the basis of file management in OpenBSD. A unique *vnode* is allocated to each active file and directory on the system.

The next call in *main()* is to *initclocks()* to establish the real time and statistical clocks. The subsequent three function calls handle initialization of structures included for UNIX System V-like capabilities. These include creating support for shared memory, semaphores and message queues as System V does. The following lines attach all the pseudo devices by first ensuring that the random device is attached, then calling the attach function for each pseudo device. Once this is has been completed the function *swcr_init()* is called to initialize the data structures needed for cryptographic support.

Initialization of the various networking protocols supported by the system is handled by the following section of code. The initialization routines are bracketed by calls that raise the interrupt priority level to that of *IPL_VM*, then return the priority level to the prior value it held. This effectively blocks the receipt of network traffic until the proper subsystems are initialized. The intervening three function calls set up a timer needed for minding the network interface, define various communication domains for use by system, and associate the domains with the various interfaces on the system.

A few optional items are handled by the next few calls, namely enabling kernel profiling and setting up support for ProPolice, a buffer overflow detection extension to *gcc*. The next function call, *init_exec()*, defines the maximum executable header size that will be possible for the system, based on the formats that will be supported such as ELF (Executable and Linkable Format), COFF (Common Object File Format) and a.out.

At this point, the timers that are used for scheduling are enabled, which is done within *scheduler_start()*. The following function within *main()*, *dostartuphooks()*, executes all functions found in the *startuphook_list* queue. The routine *startuphook_establish()* can be used to add a function to this queue which will then be run at startup. This feature is used by many device drivers in their attach functions when there are operations that must be done but cannot be run at the time the attach routine is called for various reasons. The actual number of functions that this involves may vary from release to release. In the 3.4 release four functions were defined to be on this queue.

The next few steps take care of configuring the root and swap devices, by running checksum algorithms over them and assigning major and minor device numbers. Once the root device has been configured, it is mounted via a call to *vfs_mountroot()*. The vnode structure allocated for the root directory is then retrieved and the current directory of this process is set to this vnode value. Once all of this file system management has been taken care of, Process 0 completes initialization of the virtual memory subsystem by calling *uvm_swap_init()*. This function initializes the swap system structures, thus it had to be called after file system initialization since it relies on the ability to work with *vnodes* from the recently mounted file system.

At this point, some needed accounting takes place in terms of time values. The start time of this process, Process 0, is updated to the current time and the length of time that the process has been running, the *p_rtime* value within the *proc* structure, is reset to zero.

Enough initialization has finally been done to allow the creation of other processes to begin. It is essential that Process 0 is properly established before any attempts to fork other processes are made. When a new process is forked, the call copies much of the information needed by the new process from the process structure of the parent process. Incorrectly setting the parameters of Process 0 would allow erroneous context information to propagate to all child processes.

There are two kinds of child processes that are created by Process 0. To create Process 1, which will run the *init* program, a call to *fork1()* is made. To create the other six kernel processes needed, a call to *kthread_create()* is used instead. This latter function is basically a specialized version of *fork1()*, as delving into the implementation reveals that a call to *fork1()* is made within *kthread_create()*. Similarly, the fork system call is implemented via a call to *fork1()*. As a kernel function, *fork1()* can be called directly only by kernel code or called indirectly from user space using the *fork* system call. It is instructive to compare the differences between how *fork1()* is invoked when creating Process 1 and how it is invoked within the *kthread_create()* function. The function prototype for *fork1()* reads as follows:

```
    int fork1(struct proc *p1, int exitsig, int flags, void *stack,
size_t stacksize, void (*func)(void *), void *arg, register_t *retval)
```

The *fork1()* function provides ten different flag values that are combined to create the third argument *flags* which defines many of the most important attributes of the new process to be created. The *fork1()* call found within *kthread_create()* is as follows:

```
    fork1(&proc0,  0,  FORK_SHAREVM|FORK_NOZOMBIE|FORK_SIGHAND,  NULL,
0, func, arg, rv);
```

The flags used in the third argument above indicate that the virtual memory space will be shared with the parent, Process 0, as will the signal actions and signal state. Also defined by the third argument, *flags*, is the fact that the exit status should not be left for the parent to wait and receive. To contrast, the call to *fork1()* used to create Process 1 is as follows:

```
    fork1(p, SIGCHLD, FORK_FORK, NULL, 0, start_init, NULL, rval)
```

The only flag used in this instance, FORK_FORK, indicates to kernel accounting mechanisms that this *fork1()* should be counted as a fork system call, and not a *vfork()* or *rfork()* system call. Thus the reason for creating Process 1 differently becomes clear. Since it will become a user space process, Process 1 should not be sharing memory nor signal structures with Process 0. Therefore it cannot be created by a call to *kthread_create()* which shares each of those structures by default. Instead a custom call to *fork1()* is required in order to properly initialization Process 1. The processes that are created using the *kthread_create()* call include the following:

1. A "pagedaemon" process to handle page swapping for the virtual memory subsystem,

2. A "reaper" process to free the resources still allocated to dead processes,

3. A "cleaner" process to clear out dirty buffers found in the BQ_DIRTY buffer queue,

4. An "update" process for synchronizing the file systems,

5. An "aiodoned" process for handling completed asynchronous I/O operations,

6. A "crypto" process for handling crypto requests made by user processes,

A final few additional kernel threads are created by the call to *kthread_run_deferred_queue()*. This function creates additional kernel threads as specified in the deferred request queue. During initialization when full support for forking processes was not yet in place, driver or file system code that needs to create a kernel thread can add a request to this queue via a call to *kthread_create_deferred()*. When *kthread_run_deferred_queue()* is called, that function processes the deferred queue, creating each of the kernel threads that were requested. The hierarchy of processes created on a typical OpenBSD system is shown below in Figure 6.

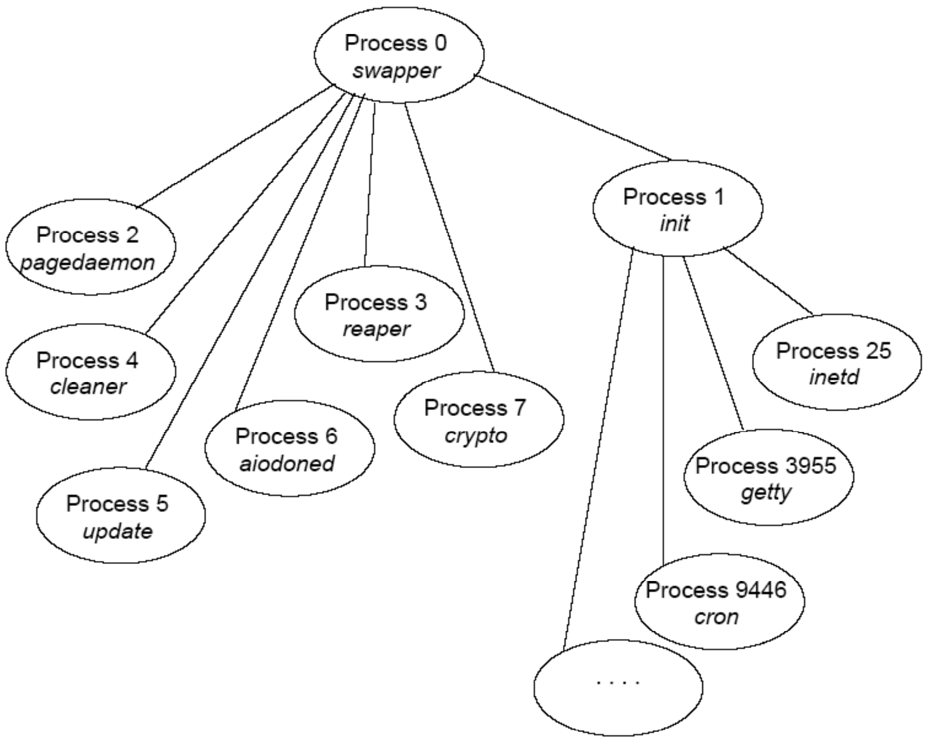


Figure 6. Process Hierarchy within OpenBSD

With all kernel threads running only a few finishing touches are needed on the system. The random number generator is seeded and the generation of process identification numbers is set up such that each new process will be given a larger pseudo-random number than the previously created process. With these final housekeeping chores taken care of Process 0 can finally enter the main loop it executes by calling *uvm_scheduler().* This function has Process 0 continually check for processes that are in a runnable state but not resident in memory and swaps them in. Control will never return to the *main()* function from this call. Once this call has been made, one could say the operating system is truly running.

At this point, it is important to note that Process 1 will go on to complete its initialization tasks after being forked from *main()* by Process 0 via a call to *fork1().* Control for Process 1 moves first to the function *start_init(),* after being forked. This function checks to make sure a console device has already been properly initialized. Then it tries to execute each of the programs found on the *initpaths* list in turn. This list contains the potential candidates for an *init* program. Typically, a valid *init* program can be found at */sbin/init* in the file system. If this fails, two other possibilities are contained in the *initpaths* list. When a valid binary is found, this binary is executed by Process 1 as the *init* program that will handle the remaining initialization that must be done within user space. With all necessary processes now running, initialization is complete.

D. SUMMARY

While the details of how initialization is implemented in OpenBSD quickly become complex, the overall architecture is straightforward. The boot loader is responsible for establishing much of the hardware-specific environment that is required. Once the kernel gains control, it immediately finishes hardware-specific tasks in a block of assembly language code, then transfers control to a high-level initialization function, *main()*. From within *main()*, the various initialization functions that have been defined by each module are then called. The final major step is to fork the various other processes needed to support the operating system. To provide some contrast to how OpenBSD handles initialization, the next chapter will analyze how the Linux operating system implements its initialization sequence.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. LINUX INITIALIZATION

A. INTRODUCTION

An overview of the initialization of the Linux kernel will be given here, with Linux 2.4.26 taken as an example. All references to source code file paths are relative to the Linux root source directory, typically named *linux-[version number]*. Much of this discussion relies heavily on [BOV02], a text that details the internals of the Linux kernel. Figure 7, provides an overview of the execution sequence during the boot process of the Linux kernel.

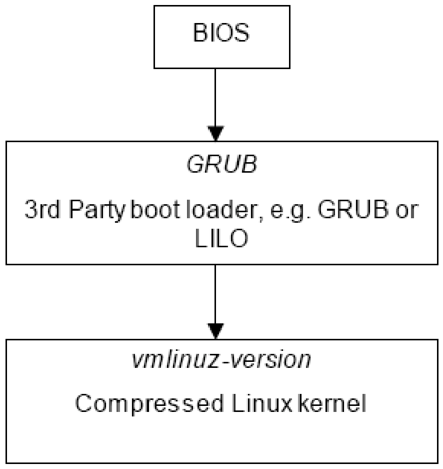


Figure 7. Linux Logical Execution Sequence from Boot to Runtime

B. ASSEMBLY LEVEL KERNEL INITIALIZATION

After the chosen boot loader has run, it then loads the Linux kernel image, typically named *vmlinuz-[version number]* for a compressed kernel image and *vmlinux-[version number]* for an uncompressed image. A compressed kernel image will have the Linux boot loader, found in */arch/i386/boot/bootsect.S*, located at the very beginning of the image. An uncompressed kernel image is simply an ELF (Executable and Linking Format) executable [TIS95]. The discussion below traces the execution flow when a compressed kernel image is being used. After the boot loader provided runs, execution begins at the start label in the assembly code file */arch/i386/boot/setup.S*. Some of the operations performed by this code include:

1. Reinitializes all hardware, since Linux does not rely on the BIOS to do this properly.

2. Ensures interrupts are disabled

3. Sets up a provisional Global Descriptor Table (GDT) and a provisional Interrupt Descriptor Table (IDT)

4. Reprograms the Programmable Interrupt Controller (PIC)

5. Re-maps the 16 IRQ lines from 0 thru 15, the BIOS assigned values, to 32 thru 47.

6. Switches from real mode to protected mode memory addressing.

The last line executed in this file is a jump to an assembly function called *startup_32()*, which performs additional initialization. There are actually two *startup_32()* assembly functions in the Linux kernel. The first *startup_32()* function to be executed is located in */arch/i386/boot/compressed/head.S*. The reason that these duplicate names do not cause any naming conflicts is because each function is reached by jumping to a physical address, rather than via a call to a label. The existence of both versions is an artifact of when there was a single *head.S* file, before compression was implemented for the kernel image. The first Linux release, version 1.0.0, had a single file */boot/head.S*, which performed all of the *startup_32()* actions. This file was moved to */arch/i386/boot/head.S* with the 1.1.45 release. With the 1.1.76 and 1.1.77 kernel releases, support for a compressed kernel was added. This resulted in the *startup_32()* functionality being split into two files, located at */arch/i386/boot/compressed/head.S and /arch/i386/kernel/head.S.*, yet each file retained a function called *startup_32()*, as the functions were never given different names. Part of what makes this convention confusing is that these two instances of *startup_32()* are executed via jumps to the same physical location rather than as two calls to their respective labels.[2] What takes place is that at the end of the initial assembly code in */arch/i386/boot/setup.S* a jump to offset

[2] If the kernel has not been loaded "high" in memory. See [BOV02] for additional detail.

0x100000 in segment __KERNEL_CS is called. This is the address where the version of *startup_32()* found in */arch/i386/boot/compressed/head.S* is located. But as part of the decompression routine, the entry point of the decompressed kernel, *startup_32()* in */arch/i386/kernel/head.S,* is relocated to that same address, 0x10000. Thus at the end of *startup_32()* found in */arch/i386/compressed/head.S* a jump to 0x100000 is called to reach the *startup_32() found* in */arch/i386/boot/head.S.* Thus while it appears from the code that these two jumps to the same physical location 0x100000 would be repetitive, they in fact are not.

The first instance of *startup_32(),* found in */arch/i386/boot/compressed/head.S,* performs the following:

1. Initializes the segmentation registers

2. Sets up a provisional stack

3. Decompresses the kernel image and locates it at address 0x10000

4. Jumps to the *startup_32()* function in the decompressed kernel, implemented in */arch/i386/boot/head.S*, which is reached by jumping to the physical address 0x100000.

In the second version of *startup_32()*, located in */arch/i386/kernel/head.S* the initialization sequence is continued. Figure 8 below details the flow of execution from this second *startup_32()* function onward through initialization, highlighting important tasks. This latter function's main job is to set up an environment within which the first process can execute. This includes the following:

1. Initializes the segmentation registers with their final values.

2. Sets up the Kernel Mode stack for Process 0.

3. Initializes the provisional kernel Page Tables

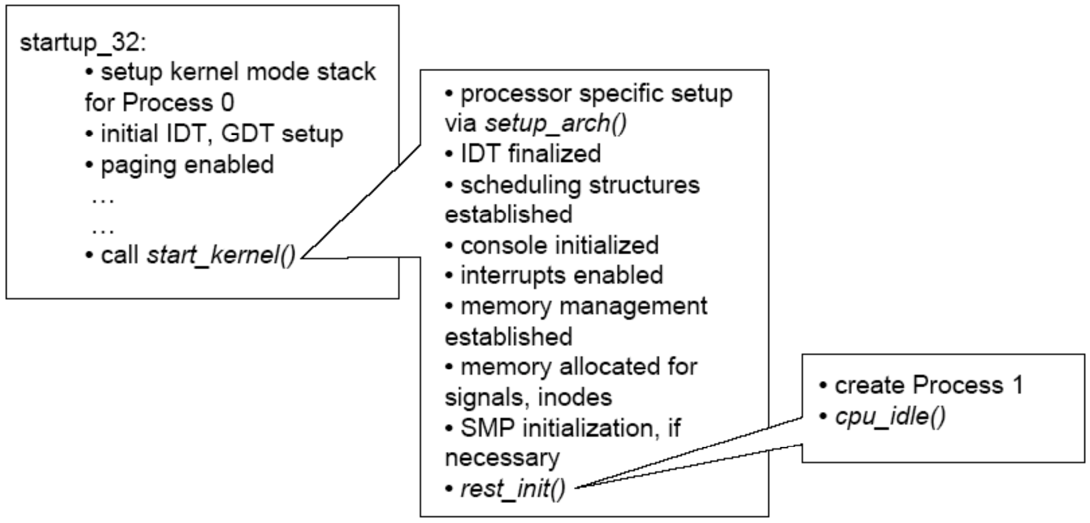
4. Stores the address of the Page Global Directory in the cr3 register, and enables paging by setting the PG bit in the cr0 register.

5. Fills the bss segment of the kernel with zeros.

6. Invokes *setup_idt()* to fill the IDT with null interrupt handlers.

7. The first page frame is loaded with the system parameters learned from the BIOS and the parameters passed to the operating system from the boot loader.

8. Loads the gdtr and idtr registers with the addresses of the GDT and IDT tables.



startup_32:
• setup kernel mode stack for Process 0
• initial IDT, GDT setup
• paging enabled
…
…
• call *start_kernel()*
• processor specific setup via *setup_arch()*
• IDT finalized
• scheduling structures established
• console initialized
• interrupts enabled
• memory management established
• memory allocated for signals, inodes
• SMP initialization, if necessary
• *rest_init()*
• create Process 1
• *cpu_idle()*

Figure 8. Execution Path within *vmlinux-[version]*, the Uncompressed Linux ELF Executable, During Initialization

C. HIGH LEVEL KERNEL INITIALIZATION

After completing these tasks, control jumps to the *start_kernel()* function. This function's role is the final initialization of all kernel components and structures. Since it touches on nearly all structures, it is instructive to discuss the important functions that *start_kernel()*, defined in */init/main.c*, performs.

- The processor specific initialization details are performed via *setup_arch()*, such as initialization of page tables and descriptors using *paging_init()*.
- The final initialization of the IDT is performed by invoking *trap_init()* and *init_IRQ()*.
- Data structures needed by the kernel for scheduling are created by *sched_init()*.

- Two of the four softirq types that are used for deferrable kernel functions are registered in the softirq array by invoking *softirq_init()*.
- The time and data used by the system are initialized in *time_init()*.
- The console device is initialized here using *console_init()*, before PCI handling is enabled. This ensures error messages will be reported to the console as soon as possible.
- The slab allocator is initialized by the function *kmem_cache_init()* and *kmem_cache_sizes_init()* which follows a few lines later.

Interrupts are enabled on line 392 via a call to *sti()*, a wrapper for the assembly level instruction *sti* that sets the IF flag in the EFLAGS register. With this bit set, "the processor begins responding to external, maskable interrupts after the next instruction is executed." [INT03-2]. This is done after *trap_init()* and *init_IRQ()* have been called to fill the IDT. While any person familiar with assembly would recognize that interrupts have been enabled, much of the enabling and disabling of interrupts found in the rest of the kernel becomes quite complex.

Linux provides a large number of macros that serve as wrappers for the *cli* (interrupt disable) and *sti* (interrupt enable) instructions. What makes this even more complex is that these macros are then used inside of other macros, many layers deep, such that to track down the fact that a macro disables interrupts requires tracking through many levels of macro definitions. For example, the following chain of *#define* statements can be found by tracing back the definition of *wq_write_lock_irq()*:

In */include/linux/wait.h*:

```
#define wq_write_lock_irq     spin_lock_irq
```

In */include/linux/spinlock.h*:

```
#define spin_lock_irq(lock)

      do { local_irq_disable(); spin_lock(lock); }

      while (0)
```

In */include/asm-i386/system.h*:

```
#define local_irq_disable()   __cli()

#define __cli()   __asm__ __volatile__("cli": : :"memory")
```

The degree to which such *#define* statements are nested adds a layer of complexity to the handling of interrupts throughout the kernel.

Execution resumes in *start_kernel()*, with interrupts enabled, with the function *calibrate_delay()*. It calculates an estimate of how many times per second a short loop can be executed by the CPU. Mainly used by device drivers, it provides a way for a driver to estimate how long it should busy-wait when waiting on information. The final few functions take care of the following tasks:

- The page frames are readied for use by the *mem_init()* function, which resets the reserved flag and calls *free_page()* on each page.
- The function *pgtable_cache_init()* initializes the x86 Physical Address Extension (PAE) caches on Intel processors with this feature
- The maximum number of processes allowed on the system is set to a default value in *fork_init()*
- Slab caches to support signals, the file system and memory are created by *proc_caches_init()*
- The slab caches as well as other kernel structures, such as inodes, needed to support the virtual file system are created by invoking *vfs_caches_init()*
- The buffer cache hash table for the file system is allocated in *buffer_init()*
- The page cache hash table is initialized in *page_cache_init()*
- The function *check_bugs()* identifies the CPU type and makes any necessary adjustments to compensate for known bugs in the respective chip
- If the kernel has been built for a uniprocessor system with an IO APIC, *smp_init()* initializes the APIC hardware. For a multi-processor system, this function is responsible for setting the system up to use all processors.

The final function called from within *start_kernel()* by Process 0 is *rest_init()*. This function invokes *kernel_thread()* to create the kernel thread that will become Process 1. A “kernel thread” in Linux is essentially a process that remains in kernel space, with access to kernel data structures. While execution for Process 0 remains inside the function *rest_init()*, the kernel thread just created, Process 1, proceeds to first execute the *init()* function found in the */init/main.c* file (only later will it execute the *init* binary which runs in user space). Discussion of Process 0 will resume later in this section. The following paragraphs trace the execution of Process 1.

The *init()* function that Process 1 enters performs a number of important final initialization tasks. The first call to *lock_kernel()* which ensures that none of the other kernel threads interfere with the device initialization that follows. The next function called is *do_basic_setup()*, which essentially handles all device initialization. It will be instructive to delve into execution of this function and take note of how support for networking is initialized in a series of steps.

The function *do_basic_setup()* first defines the process it is running within, Process 1, to be the reaper of all orphaned child processes. Each class of devices has an initialization routine that subsequently gets called, such as *acpi_init()*, *pci_init()* and *sbus_init()*. After each of these functions returns *sock_init()*, the function to initialize socket support for networking devices, is called. This function *sock_init()* initializes the array containing all protocol families, *net_families[]*, by setting each pointer to a null value. It then creates the slab memory cache for use by socket structures. Finally it registers the file system type *sock_fs_type*. Since sockets are managed just as files, identified by file descriptor numbers, this is a necessary step. Once *sock_init()* returns, the function *start_context_thread()* is called, which creates a new kernel thread. This new kernel thread, *keventd*, is responsible for running the tasks that accumulate in the *tq_context* task queue, which provides for the deferred scheduling of functions that make use of blocking operations[3]. This is a need demonstrated by many I/O devices and the interrupt handlers associated with them. Figure 9 shows the process hierarchy of all

[3] This thread previously was used to run the tasks that accumulated on the *tq_scheduler* queue, but that queue was deleted in an early version of the 2.4 kernel. See comments on *schedule_task()* in *kernel/context.c* for details.

kernel threads as well as a handful of well-known user space processes such as *sshd* and *mingetty*.

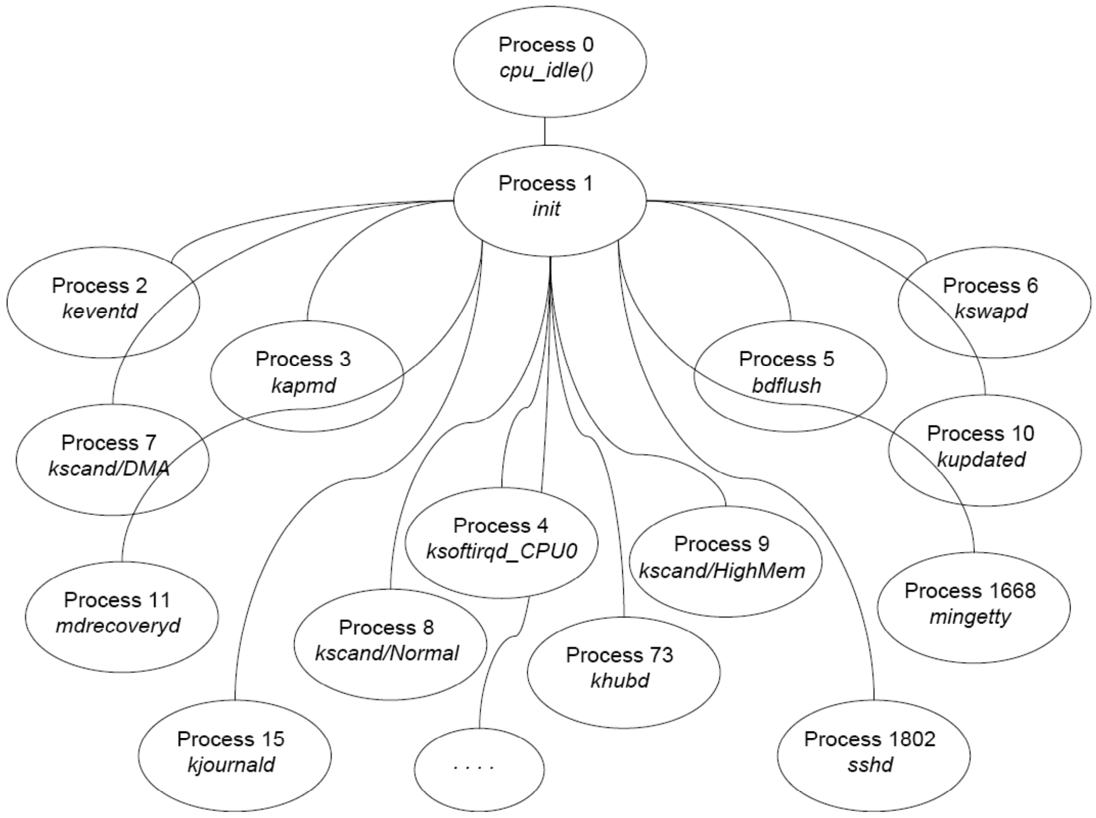


Figure 9. Process Hierarchy within Linux

Next *do_initcalls()*, the function that does most of the low-level initialization of network and other types of devices, is invoked. This function has the responsibility of calling a number of initialization functions specific to various devices and subsystems, as well as driver initialization routines. Any function wrapped with the macro *__initcall* or *module_init* will be invoked by *do_initcalls* at this point. The *__initcall* macro and the equivalent *module_init*, both defined in */include/linux/init.h*, instruct the gcc complier to place the respective function in a section named *.text.init* when linking, while the entry address of the function will be stored in the *.initcall.init* section. Thus the implementation of *do_initcalls()* involves a relatively simple loop that moves through the list of function addresses found in *.initcall.init* and calls each of them. This unique structuring of the code allows a clean seven line implementation of *do_initcall()*. Rather than having to maintain a list of each device initialization routine that must be called, this list is dynamically generated at compile time into the *.initcall.init* section.

Linux makes extensive use of the ability of gcc to place portions of code into specific sections when linking object files, provided by the *__attribute__* directive [GCC04]. Using a tool such as *readelf*, it is possible to print the various sections defined in the Linux kernel, as it is simply compiled into a regular ELF formatted binary itself. The simplified output of this command when run on the 2.4.20-30.9 kernel is shown below:

```
Section Headers:
[Nr] Name                 Type         Addr         Off          Size
[ 0]                      NULL         00000000     000000       000000
[ 1] .text                PROGBITS     c0100000     001000       153cac
[ 2] .rodata              PROGBITS     c0253cc0     154cc0       029201
[ 3] .kstrtab             PROGBITS     c027cee0     17dee0       00a03d
[ 4] __ex_table           PROGBITS     c0286f20     187f20       002910
[ 5] __ksymtab            PROGBITS     c0289830     18a830       002d68
[ 6] __kallsyms           PROGBITS     c028c598     18d598       080bec
[ 7] .data                PROGBITS     c030d200     20e200       041ec4
[ 8] .data.init_task      PROGBITS     c0350000     251000       002000
[ 9] .text.init           PROGBITS     c0352000     253000       01a28c
[10] .data.init           PROGBITS     c036c2a0     26d2a0       005dd8
[11] .setup.init          PROGBITS     c0372080     273080       0001b0
[12] .initcall.init       PROGBITS     c0372230     273230       00010c
[13] .data.page_aligne    PROGBITS     c0373000     274000       000800
[14] .data.cacheline_a    PROGBITS     c0373800     274800       001900
[15] .bss                 NOBITS       c0375100     276100       06a080
[16] .comment             PROGBITS     00000000     276100       0048d4
[17] .debug_aranges       PROGBITS     00000000     27a9d4       000028
[18] .debug_pubnames      PROGBITS     00000000     27a9fc       000022
[19] .debug_info          PROGBITS     00000000     27aa1e       0000bd
```

```
[20] .debug_abbrev         PROGBITS      00000000      27aadb        0000a9
[21] .debug_line           PROGBITS      00000000      27ab84        00008f
[22] .debug_frame          PROGBITS      00000000      27ac14        000024
[23] .debug_str            PROGBITS      00000000      27ac38        0000c2
[24] .shstrtab             STRTAB        00000000      27acfa        00012f
[25] .symtab               SYMTAB        00000000      27b264        0478c0
[26] .strtab               STRTAB        00000000      2c2b24        04ea71
```

Clearly there are quite a number of sections defined in the Linux kernel, twenty-seven to be exact, which contrasts sharply with the nine that are found in the OpenBSD kernel image. Using the *__attribute__* construction not only allows for the clean implementation of functions like *do_initcall()*, as seen above, but also allows the operating system to use memory more efficiently. All functions marked with the *__init* macro are put into the *.text.init* section when compiled and linked. The function *free_initmem()* then frees the pages in memory taken up by that section after initialization is complete, since that space is no longer needed and will not be referred to by the kernel.

Returning to examination of the code, it would not be useful to list all of the functions that are called by *do_initcall()*. Instead we will focus specifically on tracing the initialization of the networking subsystem, which is done via *inet_init()* and *af_unix_init()*, both invoked by *do_initcall()*. The job of *inet_init()* is to set up and register with the kernel the various protocols needed by TCP/IP networking. This includes setting up data structures for arp (*arp_init()*), IP, (*ip_init()*), TCP (*tcp_v4_init()*), ICMP (*icmp_init()*), IP fragmentation (*ipfrag_init()*) and multi-cast routing (*ip_mr_init()*). These routines take care of tasks like creating control sockets for both ICMP and TCP. When *af_unix_init()* is called, it registers the *unix_family_ops* protocol family, i.e., the Internet IP protocol stack, and links it into the socket structure such that a socket of this family type can be created. Other protocol families that Linux defines, and initializes via separate functions, include Novell IPX and Appletalk.

The specific device driver needed by the network devices on a given host will also be initialized by the *do_initcalls()* routine, as will all other device drivers. After all drivers have been initialized, *do_initcalls()* will return. As the last function in *do_basic_setup()*, it will also cause that function to return, placing control back inside of the *init()* function. The mount points of ramdisks and other devices, such as consoles, are determined next by invoking *prepare_namespace()*. At this point the system has completed the tasks most essential to create a working runtime environment. Now focus shifts to moving into user space to complete the initialization tasks.

The function *free_initmem()* is called to release from memory the sections of the kernel that were defined as necessary only at initialization, *.text.init, .data.init, .setup.init* and *.initcall.init*. Now unavailable to the rest of the kernel, this memory is free and may be reused. As no more operations will be performed that are sensitive to timing and race conditions, this thread no longer needs to ensure it has exclusive access to kernel data structures. Therefore Process 1 is next able to release the kernel lock it obtained upon entry to the *init()* function, by calling *unlock_kernel()*. Also, to ensure it does not inadvertently continue to have access to kernel data, the thread releases files that are currently being shared.

The final step done in kernel mode is opening the default console to use as standard input (file descriptor 0). The function *dup()* is then called twice on file descriptor 0 to allow the usage of the same console for standard output (file descriptor 1) and standard error (file descriptor 2). Finally *run_init_process()* is called, which invokes *execve()* on the argument it is passed. An attempt is first made to execute the binary that was passed to the kernel as an argument. If this fails, it tries to execute, in order, */sbin/init*, */etc/init* and */bin/init*. As a last resort the kernel attempts to execute the */bin/sh* shell as the Process 1 binary, in an attempt to provide an interface for fixing the malfunctioning system.

Once the *init* binary is executed, it begins executing each of the scripts found in the various runlevel directories. This can vary slightly with the Linux distribution, since at this point the *init* binary is executing outside of the kernel in user space. The Linux kernel does not define, nor does it compile a specific *init* binary to be run. Instead this is provided by the maker of a specific distribution of the Linux operating system, e.g., RedHat or Debian. As the user space initialization is outside the scope of the discussion here, the details of the *init* binary will not be explored.

To complete this discussion, the rest of Process 0 execution should be traced. After forking Process 1 while inside the function call *rest_init()*, Process 0 releases locked kernel resources, via a call to *unlock_kernel()*. It then sets a flag to indicate that the current process, i.e., itself Process 0, should be rescheduled. A call is then made to the function *cpu_idle()*. Once inside this function control will not return. The function consists of a low-priority idle loop where Process 0 sits waiting for other processes to request rescheduling. When such a request is made, it is Process 0 that will call the scheduler.

At this point, with Process 0 in an idle loop and Process 1 executing the *init* binary, kernel-level initialization of Linux is complete.

D. SUMMARY

Similar to OpenBSD, initialization of Linux can be divided into two main phases, excluding the BIOS—the boot loader phase and the kernel initialization phase. While both operating systems must employ assembly language code to handle tasks like probing memory or setting up operating system controlled registers, a major difference is where this assembly language code resides. OpenBSD is distributed with an integrated boot loader that moves the processor into protected memory mode. Utilizing a boot loader that handles these assembly level details allows the kernel entry point to begin with a much more robust environment, including memory protection. Therefore the amount of assembly code needed within the OpenBSD kernel has been minimized. On the other hand, the Linux kernel does not include any multi-stage boot loader code. It typically relies on the existence of third-party boot loader programs, and does not assume that the processor is put into protected memory mode. Instead the Linux kernel moves the processor into protected mode itself, requiring the kernel to include the assembly language code to establish protected memory mode. This means that overall, the Linux kernel must execute a larger amount of assembly code than OpenBSD before reaching its high-level initialization function, *start_kernel()*.

The *start_kernel()* function in Linux resembles the *main()* routine of OpenBSD to a large extent. Both are called after the CPU has been put into protected mode. Code that requires direct access to registers or flat memory addresses has already been executed. Both *start_kernel()* and *main()* are processor independent and all code within these two functions is written strictly in C. Any machine-dependent functions are hidden within the functions called by these two high-level routines. Each module or subsystem within both operating systems defines an initialization function that is executed by the high-level initialization routine. In both OpenBSD and Linux these initialization functions are defined as part of the modules themselves, not as part of a separate initialization module.

While the details of implementation are slightly different, the overall architecture used to bring both operating systems up to a running state is essentially the same. Any necessary processor-specific items are first handled by a block of assembly code. Once these low-level details are sufficiently handled, for example moving the processor into protected mode on an Intel x86 CPU, control moves to a high-level machine-independent initialization function. This function completes initialization, including the creation of any additional processes needed by the system.

With this detailed understanding of how initialization is handled differently by two UNIX-like operating systems, we can move to a discussion of the role that the initialization sequence plays in the security of an operating system. As the "security" of a system can only be assessed relative to a security policy, it is not possible to make sweeping statements about the "security" of the initialization routine of either OpenBSD or Linux. Instead the question will be approached by looking at the possible threats that may exist in a given environment. While only a minimal set of threats may be present in a given environment, requiring only a minimal level of "security," a different environment might require a greater number of countermeasures be in place. The different types of threats that must be considered, and their relationship to system initialization will be presented in the following chapter.

THIS PAGE INTENTIONALLY LEFT BLANK

V. THREAT ASSESSMENT

A. INTRODUCTION

Having explored the way that the initialization sequence is implemented in two operating systems in detail, it would be interesting at this point to be able to make a statement about the security of this part of the operating system. The preceding analysis is of interest due to the lack of documentation and lack of community understanding about the initialization sequence. It is exactly this lack of documentation that should lead one to be suspicious about the level of scrutiny that this section of the code has received, and therefore suspicious of its security.

But any effort to define whether a system is "secure" or not must be done relative to a security policy. Without a clear idea of what "secure" is supposed to look like, there is no possible way to know when it has been reached. Such a security policy is in turn designed to reduce the level of risk posed by an operational system. A common definition of residual risk is threats multiplied by vulnerabilities minus countermeasures. If there are neither threats to, nor vulnerabilities within, the initialization sequence, no security analysis is required. In that case, the initialization routine, by definition, would not be the source of any risk. To provide proper context for a discussion about the security and assurance that must be provided by the initialization sequence, an understanding of both threats and vulnerabilities must be developed. This chapter focuses on developing the threat picture, using documents developed for the Common Criteria as a guideline.

To develop an understanding of the vulnerability picture, parallel to the threat picture developed in this chapter, different tactics are available. A detailed and robust vulnerability analysis would require a thorough code audit. The expertise and time required to undertake such an audit is beyond the scope of this thesis. Instead an examination of the design and structure of the code at a high level will be performed. By analyzing the degree to which this code exhibits the use of good security engineering practices, it should be possible to gauge the likelihood of existing vulnerabilities.

What will be presented in this chapter is a review of threats that could be considered applicable to the initialization sequence. The first half will focus on the threats that must be addressed in order for an operating system to be certified under the Common Criteria to operate in an environment requiring "medium robustness" [SLP03]. A determination will then be made as to which threats, if any, are applicable to the initialization routine, using the OpenBSD operating system as a reference. This will provide a foundation for asserting whether scrutiny of the initialization code is necessary and appropriate for medium robustness systems. If a threat is determined to be applicable, close attention will be paid to what can be done to mitigate this threat in the final recommendations chapter. The discussion of threats addressed by medium robustness operating systems will be followed by a discussion of the threats that must be addressed by an operating system in an environment requiring high robustness. In a similar manner, it will be determined which of these threats are applicable to the initialization sequence. This will provide a basis for understanding the measures that must be in place to address high assurance initialization requirements.

B. SOURCE OF MEDIUM ROBUSTNESS THREAT FRAMEWORK

In order to couch this threat discussion within a recognized framework, a number of Common Criteria Protection Profile documents will be used for reference. As a thorough description of the Common Criteria is beyond the scope of this thesis, a brief overview follows.

1. The Common Criteria

In the 1980s the U.S. government developed the Trusted Computer System Evaluation Criteria (TCSEC) standard, which came to be known simply as the "Orange Book." It outlined a methodology for evaluating the assurance level of an operating system. Subsequently a number of other countries each developed their own evaluation methodology. It was soon recognized that much of this effort was redundant and created large obstacles to evaluation. Due to the expense a vendor had to go to in order to have a product evaluated, it was simply not feasible for a company to evaluate a product under multiple schemes. A move towards a more global standard produced the US Federal Criteria, from which today's Common Criteria was created. Similar to the TCSEC, the Common Criteria provides a framework for evaluating the assurance level of hardware and software products. Unlike the TCSEC, it is not U.S. specific but has instead been developed as a joint effort between a number of nations.

The main philosophical difference between the TCSEC and Common Criteria lies in the difference between functional requirements and assurance requirements. While the TCSEC tied the two together, the Common Criteria considers each category of requirements separately. Therefore if a large number of functional security requirements are needed for a system, this does not necessarily mean that the system must be built with a high assurance level.

The Common Criteria evaluation framework defines a number of documents that are used to guide the evaluation process. In order for an evaluation to be done, a Target of Evaluation (TOE) must first be defined. This is a statement of what will be included in the evaluation, what pieces of software are within the scope of the evaluation as well as what type of hardware it will run on. Based on the TOE, a Security Target (ST) is usually generated which documents how the TOE meets the necessary requirements for the functional and assurance requirements it intends to fulfill. To avoid writing an ST from scratch for each new product evaluation, the notion of a Protection Profile (PP) was developed. The purpose of a Protection Profile is to serve as a template for writing a Security Target. In other words, a PP would be developed that defines the minimal requirements for an operating system that provides medium robustness. The requirements put forth in this PP can then be used to generate a Security Target document describing a specific operating system to be evaluated.

While a Security Target does not need to be made public, as it may contain trade secrets, a Protection Profile must be made openly available. This is one of the main reasons that a vendor might choose to bypass generation of a Protection Profile and move straight to product evaluation using only a Security Target document.

2. The Protection Profile

Within a Protection Profile, a section is devoted to listing the threats that must be addressed in order for a TOE to be successfully evaluated against that PP. It is this threat section, from various PPs, that will be drawn on for discussion here. The validated "Protection Profile for Single-level Operating Systems in Environments Requiring Medium Robustness" (SLOS PP) can be found on the Common Criteria web pages [CSL03]. This SLOS PP sets forth requirements that would lead to an EAL 4+ rating. A number of US produced Common Criteria Protection Profiles, including the SLOS PP used here, define and make extensive use of “robustness” classes, as illustrated in the following diagram. Figure 10 gives the reader a sense of what is meant by “medium robustness,” as used in the SLOS PP title. In this PP, it is assumed that adequate physical security is being provided for the TOE, based on the value of the data being protected by the TOE. Therefore the common threat of physical compromise, and related issues such as booting into single-user mode, are not included in the threat list for this PP.

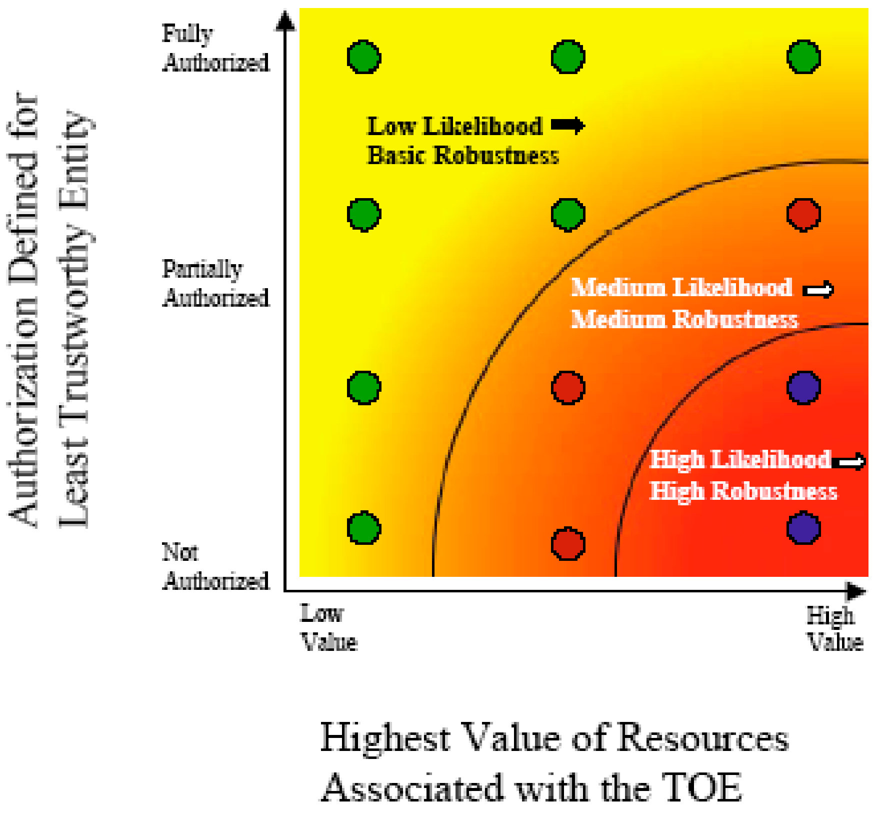

Authorization Defined for Least Trustworthy Entity
Fully Authorized
Partially Authorized
Not Authorized
Low Likelihood ➡ Basic Robustness
Medium Likelihood ⇨ Medium Robustness
High Likelihood ⇨ High Robustness
Low Value
High Value
Highest Value of Resources Associated with the TOE

Figure 10. Robustness Requirements based on Compromise Likelihood (From: [SLP03])

The overarching question when examining these threats in this section is: does the ability to mitigate these threats rely on specific measures being taken in designing and implementing the initialization routine? If so, then additional care must be taken to ensure that proper countermeasures to these threats are in place. The final chapter will describe exactly what kind of “additional care” should be taken, and will be in the form of specific recommendations to be followed when designing and implementing the initialization sequence of an operating system.

C. RELEVANCE OF MEDIUM ROBUSTNESS THREATS TO INITIALIZATION SEQUENCE

The following discussion is based on the threats listed in “Protection Profile for Single-level Operating Systems in Environments Requiring Medium Robustness.” [SLP03] Since none of those threats makes specific mention of the initialization sequence, some examination is needed to determine which threats do in fact involve the initialization sequence. The intent below is to explore whether countering a given threat would impose requirements upon the initialization routine. To make this determination, the OpenBSD operating system will be used as a reference. The scope will be strictly limited to consideration of the kernel initialization sequence, excluding any boot loader activity.

For those threats that are relevant to the initialization sequence, it will be important to ensure that the concluding chapter on recommendations encompasses countermeasures to these threats. Those threats that are tangential to the initialization sequence will not be addressed beyond this section. The table below, Table 1, lists each threat found in the SLOS PP and whether it could be considered relevant to the initialization routine of the OpenBSD operating system. The rationale behind this relevance determination follows.

Relevant	Threat	Description
Y	ADMIN_ERROR	An administrator may incorrectly install or configure the TOE resulting in ineffective security mechanisms.
Y	ADMIN_ROGUE	An authorized administrator’s intentions may become malicious resulting in user or TSF data being compromised.
Y	AUDIT_ COMPROMISE	A malicious user or process may view audit records, cause audit records to be lost or modified, or prevent future audit records from being recorded, thus masking a user’s action.

Relevant	Threat	Description
N	CRYPTO_ COMPROMISE	A malicious user or process may cause key, data or executable code associated with the cryptographic functionality to be inappropriately accessed (viewed, modified, or deleted), thus compromising the cryptographic mechanisms and the data protected by those mechanisms.
N	EAVESDROP	A malicious user or process may observe or modify user or TSF data transmitted between physically separated parts of the TOE.
N	MASQUERADE	A malicious user, process, or external IT entity may masquerade as an authorized entity in order to gain unauthorized access to data or TOE resources.
Y	POOR_DESIGN	Unintentional or intentional errors in requirements specification or design of the TOE may occur, leading to flaws that may be exploited by a malicious user or program.
Y	POOR_ IMPLEMENTATION	Unintentional or intentional errors in implementation of the TOE design may occur, leading to flaws that may be exploited by a malicious user or program.
Y	POOR_TEST	Lack of or insufficient tests to demonstrate that all TOE security functions operate correctly may result in incorrect TOE behavior being undiscovered thereby causing potential security vulnerabilities.
N	REPLAY	A user may gain inappropriate access to the TOE by replaying authentication information, or may cause the TOE to be inappropriately configured by replaying TSF data or security attributes.

Relevant	Threat	Description
N	RESIDUAL_DATA	A user or process may gain unauthorized access to data through reallocation of TOE resources from one user or process to another.
N	RESOURCE_ EXHAUSTION	A malicious process or user may block others from system resources (i.e., system memory, persistent storage, and processing time) via a resource exhaustion denial of service attack.
N	SPOOFING	A malicious user, process, or external IT entity may misrepresent itself as the TOE to obtain authentication data.
N	TSF_ COMPROMISE	A malicious user or process may cause TSF data or executable code to be inappropriately accessed (viewed, modified or deleted).
N	UNATTENDED_ SESSION	A user may gain unauthorized access to an unattended session.
N	UNAUTHORIZED_ ACCESS	A user may gain unauthorized access (view, modify, delete) to user data.
Y	UNIDENTIFIED_ ACTIONS	The administrator may fail to notice potential security violations, thus preventing the administrator from taking action against a possible security violation.
Y	UNKNOWN_STATE	When the TOE is initially started or restarted after a failure, the security state of the TOE may be unknown.

Table 1. Threat Relevance (After: [SLP03])

1. Relevance of ADMIN_ERROR

An administrator may incorrectly install or configure the TOE resulting in ineffective security mechanisms.

This threat does touch on the initialization phase of OpenBSD. While the installation of the system, as mentioned above, is not a concern of the initialization sequence, the configuration is. During overall system bootstrap there are two points where configuration inputs are allowed. The OpenBSD boot loader allows a user with physical access to the machine to specify boot options. Yet since this is handled by the boot loader and not the initialization sequence, it is outside the scope here and will not be considered.

OpenBSD also provides a compile time option which allows a user to specify kernel configuration options at initialization. The prompt for kernel configuration options is generated by the function *user_config()*, called within *cpu_startup()* which is clearly part of the initialization sequence as it is called within the high-level initialization function *main()*. The configuration inputs allowed primarily deal with device management and thus do not appear directly related to any security mechanisms. Yet this is a possible vector which could allow an administrator to introduce incorrect configuration and unintentionally give a malicious device access to the system.

2. Relevance of ADMIN_ROGUE

An authorized administrator's intentions may become malicious resulting in user or TSF data being compromised.

This is a significant threat, relevant to the initialization sequence in a way similar to the threat above. The above threat dealt with accidental misconfiguration, while this threat covers intentional misconfiguration or damage. While the possibility of a malicious administrator misconfiguring the system exists, using the kernel configuration mechanism discussed above, the only other time an administrator could impact the system, short of pulling the power cord, would be after the initialization routine has run. Once the system is up and running, a malicious administrator could also recompile the kernel, after making changes to any part of the system, including the initialization sequence. This is a clear threat that does involve the initialization sequence.

3. Relevance of AUDIT_COMPROMISE

A malicious user or process may view audit records, cause audit records to be lost or modified, or prevent future audit records from being recorded, thus masking a user's action.

Since the initialization sequence provides an audit trail of its activity during boot, this should be protected in the same way that other audit records are, therefore this threat is quite relevant. A log of initialization activity is written to kernel memory during the initialization sequence, which is then saved to a file at */var/run/dmesg.boot* by the */etc/rc* script that is run as part of user space initialization. Therefore, while the initialization sequence is responsible for properly generating and writing the audit to memory, proper preservation of the audit trail also depends on other aspects of the system. Yet, this threat is ultimately of concern for our discussion here due to the role that the initialization routine plays in audit generation.

4. Relevance of CRYPTO_COMPROMISE

A malicious user or process may cause key, data or executable code associated with the cryptographic functionality to be inappropriately accessed (viewed, modified, or deleted), thus compromising the cryptographic mechanisms and the data protected by those mechanisms.

This is a threat isolated within a single module of the system, namely the cryptographic functionality, assuming the initialization sequence properly handled initialization of this module. If the initialization sequence had not done so, this would fall under the threat POOR_DESIGN, discussed later. It is the cryptographic module alone, and the associated file system and memory protections, that are responsible for protecting the keying data as well as any related functions. This threat can be considered not applicable to OpenBSD initialization.

5. Relevance of EAVESDROP

A malicious user or process may observe or modify user or TSF data transmitted between physically separated parts of the TOE.

Typically a single OpenBSD system would not have various pieces of the operating system physically separated. The only time this case would arise is when doing remote boot. For this discussion the assumption will be that remote boot functionality is not being used, therefore this threat will not be taken into consideration by the initialization sequence.

6. Relevance of MASQUERADE

A malicious user, process, or external IT entity may masquerade as an authorized entity in order to gain unauthorized access to data or TOE resources.

Since there is no concept of authorization or permissions at the initialization stage, outside of the authority to have physical access, this threat is not applicable to the initialization sequence. As stated in the introduction to this section, physical access is assumed to be properly assured. With this assumption, such issues as single-user mode do not need to be considered, as physical access is required to exploit this vulnerability.

7. Relevance of POOR_DESIGN

Unintentional or intentional errors in requirements specification or design of the TOE may occur, leading to flaws that may be exploited by a malicious user or program.

Every part of the OpenBSD operating system would be susceptible to flaws introduced by a faulty design. Thus this threat is quite relevant to initialization. The larger question is perhaps, what kind of effect could such a flaw introduce? A major design flaw in the initialization code would certainly be detected once the system reaches the implementation phase, as any significant omission in the initialization code would produce quite significant and noticeable errors in the functionality of the rest of the system. More insidious than an omission would be the introduction of extra functionality by a malicious developer. This threat will be discussed more fully under the next heading.

8. Relevance of POOR_IMPLEMENTATION

Unintentional or intentional errors in implementation of the TOE design may occur, leading to flaws that may be exploited by a malicious user or program.

Again, the potential exists for any part of the OpenBSD system to be affected by implementation errors, whether intentional or not, thus this threat encompasses the possibility of a developer subverting the system by inserting extra functionality. If part of a module was unintentionally set up incorrectly by the initialization sequence, it could lead to an exploitable condition. While the initialization routine does set up all the data structures to be used by the operating system, it is not code that is typically executed repeatedly once the system is up and running. But if a malicious developer intentionally inserted a subversion artifice into the initialization routine, it would only need to run once in order to alter surrounding code such that the subversion would remain active after initialization was finished. Yet with an operating system like OpenBSD, if a developer had malicious intent and was able to insert an artifice into the initialization code, they would be equally likely to simply insert it into the runtime code. In fact this is probably a more likely scenario, since a runtime artifice could take advantage of the full capabilities of the runtime system. Therefore this threat is equally applicable to the initialization code as it is to any other part of the code base.

9. Relevance of POOR_TEST

Lack of or insufficient tests to demonstrate that all TOE security functions operate correctly may result in incorrect TOE behavior being undiscovered thereby causing potential security vulnerabilities.

An insufficient level of testing could result in incorrect behavior of the initialization routine going unnoticed. Thus this threat is relevant to the initialization routine. Within the initialization routine there exists the potential for quite subtle timing flaws that would create race conditions that would be difficult to reproduce and thus debug. Thus this threat, while equally applicable to all code, is perhaps particularly relevant to the initialization routine.

10. Relevance of REPLAY

A user may gain inappropriate access to the TOE by replaying authentication information, or may cause the TOE to be inappropriately configured by replaying TSF data or security attributes.

Since there is no concept of authentication at the initialization stage, the threat of replayed information does not exist at that level. Similarly, the only configuration information that could be replayed during boot is the kernel configuration parameters that are allowed. In order to do this kind of replay, physical access would be required. But as sufficient physical security is assumed in this analysis, as stated in the preceding section, the latter half of the threat stated above is also not applicable to the initialization sequence.

11. Relevance of RESIDUAL_DATA

A user or process may gain unauthorized access to data through reallocation of TOE resources from one user or process to another.

The very nature of volatile memory means that it is cleared when the machine is initially powered on, thus concerns about reuse of memory are not relevant from that point of view. Yet if a system were to restart from a warm boot, i.e. without cycling power but using a key sequence such as CTRL-ALT-DEL, main memory will not be cleared. This raises the question of what the initialization routine needs to do differently to handle this situation to protect against improper reallocation of memory. But if uncleared memory is accessible to the kernel code after a warm boot this is not unauthorized access, as the kernel has permission to access any part of memory in the first place. Since the kernel runs at the highest privilege level, it inherently has access to this memory. Therefore the ability of the kernel to access uncleared memory after a warm boot is not of concern. What is of concern is the ability of a user space process to access such memory. But this would be handled by the existence of functions within the memory manager and file system dedicated to object reuse which would be invoked on every allocation of memory or disk space. Therefore this threat is not relevant to initialization.

12. Relevance of RESOURCE_EXHAUSTION

A malicious process or user may block others from system resources (i.e., system memory, persistent storage, and processing time) via a resource exhaustion denial of service attack.

While the initialization sequence controls the initial data structures defined to support the allocation of system resources, it is not responsible for further management of these resources. Thus it falls within the realm of a resource-specific subsystem, e.g. the file system for files or the memory manager for new process memory space, to ensure that resource exhaustion does not lead to a denial of service attack. This threat is not relevant to the initialization code.

13. Relevance of SPOOFING

A malicious user, process, or external IT entity may misrepresent itself as the TOE to obtain authentication data.

As this threat deals with abuse of authentication mechanisms, it is not applicable to the initialization sequence since there is no concept of authentication or permission at the initialization stage for OpenBSD, outside of the authority to have physical access. As stated previously, physical security is assumed.

14. Relevance of TSF_COMPROMISE

A malicious user or process may cause TSF data or executable code to be inappropriately accessed (viewed, modified or deleted).

This threat is concerned with "inappropriate" access which implies that at the time of the access some policy defining appropriate access is in place. Since there is no concept of access control or authentication at the initialization stage, the threat of inappropriate access does not exist at the initialization stage. The design of the initialization sequence is such that it must be able to access and modify data across the system. Therefore this threat is not relevant to the initialization sequence.

15. Relevance of UNATTENDED_SESSION

A user may gain unauthorized access to an unattended session.

Since it is not possible to establish user sessions to OpenBSD until initialization is complete and runtime code is executing, the threat of unauthorized access to such a session does not involve the initialization sequence.

16. Relevance of UNAUTHORIZED_ACCESS

A user may gain unauthorized access (view, modify, delete) to user data.

This threat is concerned only with the possibility of users accessing the data of other users. As the very notion of a user who is not an administrator is only applicable once in a runtime state, this threat does not apply to the initialization sequence.

17. Relevance of UNIDENTIFIED_ACTIONS

The administrator may fail to notice potential security violations, thus preventing the administrator from taking action against a possible security violation.

The initialization sequence is involved in this threat, since it must produce an audit log of its own actions, as discussed under AUDIT_COMPROMISE above. While that threat is concerned with the compromise of audit records, the threat here addresses whether the proper indicators are being recorded in the first place. Just as for the rest of the system, care must be taken to ensure that audit of the initialization sequence results in appropriate, actionable information about possible security violations.

18. Relevance of UNKNOWN_STATE

When the TOE is initially started or restarted after a failure, the security state of the TOE may be unknown.

This final threat is applicable to the initialization routine. The state of the TOE when it is initially started is determined by the initialization routine, as well as the underlying hardware and any boot loader that executes prior to the TOE operating system. Thus examination of the initialization routine is needed in order to be able to make any statements about whether or not the state is known or unknown after startup or after a failure.

D. SOURCE OF HIGH ROBUSTNESS THREAT FRAMEWORK

The preceding analysis identified a number of threats relevant to the initialization routine of a medium robustness operating system. This section examines how the threat picture changes when a higher level of robustness is required. As seen earlier, it is common for US Protection Profile documents to define three general classes of robustness: basic, medium and high. The previous section considered only those threats addressed by a medium robustness operating system. The following section will examine some of the threats that must be considered when building a high robustness operating system.

Naturally there is a great deal of overlap between the two. Yet there are added threats that must be addressed when building a high robustness operating system. Explicit in the Common Criteria notion of robustness is the idea that data compromise is more likely when unauthorized users have access to a system handling high value data than when only authorized users have access to a system handling high value data. This gives rise to concern over issues such as covert channels. While the existence of covert channels is not considered a threat to a medium robustness system, it becomes a concern for environments where high value data could be leaked to a user with no authorized access to that data.

While the section above relied on the OpenBSD operating system to anchor the discussion, an implementation of a high robustness system that is publicly accessible is not available to anchor the discussion below. As mentioned in the introductory chapter all high assurance implementations to date have been of a proprietary nature. For this reason, this thesis has been limited to exploration of open source operating systems. While the recommendations that conclude this thesis aspire to be applicable to high assurance operating systems, the analysis in the background chapters was necessarily restricted to open source operating systems.

The threats identified in the following section have been drawn from the draft of "Protection Profile for Separation Kernels in Environments Requiring High Robustness" that has been published on the Common Criteria website [SKP04]. These are threats that are considered beyond the scope of the average commercial operating system, but essential to defining necessary protection in a high robustness system. Any threats that substantially overlap with those presented in the preceding section have been omitted. The focus below is solely on the threats that are added when one moves from an environment requiring a medium robustness system to an environment requiring a high robustness system. As above, the question when examining these additional threats is: does the initialization routine of a high robustness system need to be involved in order to mitigate these threats?

E. RELEVANCE OF HIGH ROBUSTNESS THREATS TO INITIALIZATION SEQUENCE

To establish threats that are typically addressed by high robustness operating systems, a protection profile currently in development for a separation kernel in a high robustness environment was consulted [SKP04]. Among those threats that are unique to a high robustness system, some common themes emerged. First, more subtle and technically sophisticated threats, such as covert channels, are of concern in high assurance systems. Second, a larger slice of the product lifecycle is considered to be within the scope of evaluation. For example, a high assurance system should provide a trusted delivery mechanism to ensure that the final product is not corrupted or modified between the vendor site and the client site. A final theme that runs through discussion of high assurance threats involves the notion of secure state. This idea of secure state, and proving formally that only other secure states can be reached from an initial secure state, is quite central to how high assurance systems provide a high degree of confidence in their correct function. The five threats below, selected for discussion as the only threats that did not overlap substantially with those in the previous section, touch on all three of these themes. The analysis that follows attempts to determine the extent to which these threats affect the initialization sequence of a high robustness system.

Relevant	**Threat**	**Description**
Y	ALTERED_ DELIVERY	The TOE may be modified during delivery such that the version received by the client does not match the master distribution version.
N	BAD_ RECOVERY	The TOE may be placed in an insecure state as a result of unsuccessful recovery from a system failure or discontinuity.
N	COVERT_ CHANNEL_EXPLOIT	An unauthorized information flow may occur between parts of the TOE as a result of covert channel exploitation.
N	INCORRECT_BOOT	The TSF implementation and TSF data are not correctly transferred into the TSF's execution domain.
Y	INSECURE_STATE	When the TOE is initially started or restarted after a failure, the security state of the TOE may be in an insecure state.

Table 2. Threats Addressed by High Robustness Operating Systems (After: [SKP04])

1. Relevance of ALTERED_DELIVERY

The TOE may be modified during delivery such that the version received by the client does not match the master distribution version.

This threat, appropriately identified as important for a high assurance system, does involve initialization code. Actual physical delivery of the TOE could be assured using cryptographic protocols to verify the integrity of the code being delivered. But it is equally important to ensure that on an ongoing basis, no unauthorized changes have been made to the kernel image or related applications. The ability to do this kind of verification would involve the initialization sequence, which will be discussed further in the recommendations chapter.

2. Relevance of BAD_RECOVERY

The TOE may be placed in an insecure state as a result of unsuccessful recovery from a system failure or discontinuity.

This threat appears to concern itself solely with the state resulting from invoking a recovery mechanism. Depending on the design of the system recovery mechanism, the initialization sequence may or may not be involved. This threat also overlaps substantially with the threat INSECURE_STATE covered later in this section. Therefore this threat will be considered restricted to the specific recovery mechanism in place and not relevant to the initialization sequence.

3. Relevance of COVERT_CHANNEL_EXPLOIT

An unauthorized information flow may occur between parts of the TOE as a result of covert channel exploitation.

The covert channel issue is of concern during system operation, as data classified at a high level may be leaked to a user at a lower level not authorized to see that data. In order for this to take place a program must typically be added to the system or run on top of the TOE. It is this added program that exploits a pre-existing mechanism for signaling data from a high to a low level. The ability to add or run a program on top of the TOE would only be possible once the system was up and running. The only way of adding a program to run during system initialization would be to subvert the development of this section of code. Thus, excluding the subversion threat which merits a category of its own, the covert channel threat is not relevant to the initialization sequence.

4. Relevance of INCORRECT_BOOT

The TSF implementation and TSF data are not correctly transferred into the TSF's execution domain.

This threat is concerned with the way in which the system kernel and data is moved into the running execution environment. Thus it is essentially concerned with the functionality of the boot loader being used to load and execute the kernel image. As the boot loader has been defined to be outside the scope of the analysis here, this threat will not be considered further.

5. Relevance of INSECURE_STATE

When the TOE is initially started or restarted after a failure, the security state of the TOE may be in an insecure state.

This threat involves the initialization sequence quite intimately as it is concerned with the state of the system when it has been restarted after a failure. It is the responsibility of the initialization sequence to ensure that a proper secure state has been achieved once it completes. In a high robustness system, formal analysis is used to prove that from an initial secure state only other secure states can be reached. Generally it is not formally proven that an initial secure state is achieved by the initialization sequence. Without sufficient analysis of the operating system, the reality of such an initial secure state can be questioned. Therefore this threat, which is directly relevant to the initialization sequence, must be countered using methods other than formal proof. These methods will be explored in the final recommendations chapter.

F. SUMMARY

This analysis has revealed the following eight threats to be relevant to the initialization sequence of a medium robustness operating system: ADMIN_ERROR, ADMIN_ROGUE, AUDIT_COMPROMISE, POOR_DESIGN, POOR_IMPLEMENTATION, POOR_TEST, UNIDENTIFIED_ACTIONS, and UNKNOWN_STATE. It has also revealed the following two threats to be relevant to the initialization sequence of a high robustness operating system: ALTERED_DELIVERY and INSECURE_STATE. It should be emphasized that the discussion above was intended to clarify the extent to which countermeasures to the above threats would involve the initialization sequence. The discussion did not attempt to fully address the extent to which OpenBSD counters these threats, rather simply tried to identify whether or not countermeasures are called for. OpenBSD has not been built with high assurance requirements in mind, and that is part of the reason that high assurance threats were discussed separately. So, while many of the recommendations in the concluding chapter will address the medium robustness threats explicitly, they should be equally applicable to a high robustness system.

Some of the threats outlined above, such as AUDIT_COMPROMISE, would appear to be quite testable, while others are not in any kind of practical sense. The problem with testing is that while only a small number of cases must be tested in order to ensure proper behavior on expected inputs, to test the negative requirement that no extra functionality exists requires a much more exhaustive approach. For example, one might want to test that no extra functionality existed within the TCP/IP stack implementation of a system. To do so would require testing of all $2^{12,000}$ possible packets, assuming a 1,500 byte maximum packet size imposed by Ethernet. To put this very large number in perspective, it is estimated that there are 2^{170} atoms in the earth and 2^{223} atoms in the entire galaxy [SCH96].

Therefore a threat like POOR_IMPLEMENTATION, related to elimination of extra functionality, cannot be addressed by a single countermeasure, such as sufficient testing. Instead countering such a threat requires good security engineering, amongst other measures like using well-trained and cleared developers. Due to this need, before delimiting the specific countermeasures, we will explore the realm of security engineering. The purpose of such an exploration is twofold: since good engineering plays such a key role in countering the POOR_IMPLEMENTATION threat, it should be explored further. Also, as mentioned at the beginning of this chapter, the security engineering of a system also gives an indication as to the probability that there are extant vulnerabilities in the system.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. SECURITY ENGINEERING ANALYSIS

A. INTRODUCTION

Having examined the threats relevant to the initialization code, the analysis in this chapter takes on new meaning. Initially it was driven by the desire to estimate the probability of a vulnerability existing in the initialization routine, based on the assumption that code that is better engineered will be less likely to have flaws. Based on the threats examined above, a new purpose can now also be seen. Not only is well-engineered code less susceptible to unintentional errors, it is also less likely to contain a subversion artifice. Since the discussion above led to the conclusion that subversion is one of the greatest threats to the initialization sequence of a high assurance OS, it would be appropriate to counter this threat by utilizing good security engineering principles. The term "security engineering" is used in favor of "software engineering" as traditional software engineering does not concern itself with malicious intent on the part of developers. The main goals that have driven the development of security engineering practices is the desire to be able to create understandable, transparent code that can be mapped back to the specification easily. This is exactly what is needed to counter the threat of subversion.

In this chapter, five security engineering aspects of the initialization code of the OpenBSD operating system will be examined. These aspects are: modularity, layering, cohesion, coupling, and minimization of complexity. These were chosen due to their prominence in the literature on security engineering. While a formal definition of each is given within the respective subsection, it is useful to note here that these aspects of a software project are quite interrelated, often defined in terms of one another. For example, software that is decomposed into appropriate modules will be inherently less complex than the same piece of software written as one large module. These relationships will become clearer as the discussion moves forward.

Definitions of each of the engineering principles will be presented first. The second half of this chapter will focus on how these principles apply to the OpenBSD system. While the main intent in the following sections is to focus on how the initialization code in OpenBSD has been engineered, the scope is necessarily slightly wider. Due to the nature of the initialization routine it is difficult to discuss the engineering aspects of this part of the OpenBSD system in a vacuum, as it is quite dependent on how other parts of the operating system have been implemented. For that reason, while the main thrust of the sections that follow is discussion of the initialization code, other aspects of the OpenBSD operating system will be touched on as well.

B. DEFINITIONS

1. Modularity

A definition of modular decomposition, taken from the IEEE Standard Glossary of Software Engineering Terminology [IEEE90], states that it is "the process of breaking a system into components to facilitate design and development." The goal of modularity is to hide design decisions, encapsulating them within modules [PAR72]. This allows those design decisions to change within a module without affecting the way other modules communicate with it. Examples of different ways of doing modular decomposition are illustrated elegantly in a paper by Parnas [PAR72]. In a typical operating system we may think of a module as a grouping of functions responsible for handling a specific high-level task. Thus the code implementing a specific cryptographic algorithm could be categorized as a single module. While the principles of data hiding and encapsulation are often cited as principles distinct from modularity, they will be considered under the modularity heading in this chapter.

2. Layering

Layering can be defined as "the design of software such that separate groups of modules (the layers) are hierarchically organized to have separate responsibilities such that one layer depends only on layers below it in the hierarchy for services, and provides its services only to the layers above it." [SKP04] Use of layering contributes significantly to the clarity of an implementation. This was the model used by Dijkstra for construction of the "THE" operating system [DIJ68]. His paper on development of "THE" stated that the design allowed the team to exhaustively test each layer before moving on to testing the next layer, thus proving the correctness of the entire system. Often the motivation for layering is not only software clarity but also to make the porting of software to a different architecture easier. Thus the core functionality of the kernel can be written in a machine independent way, by providing platform-independent interfaces to system resources. All communication and requests for resources are addressed to the machine-independent layer. The actual hardware dependent code that implements the functionality in a platform-specific manner is contained in another layer beneath the machine-independent layer. Each machine-independent layer then calls down to the corresponding hardware dependent layer to actually handle the needed operations. Only a small subsection of the code will then require modification when the software is ported to a different architecture.

3. Cohesion

Another aspect of software design that is useful to examine is cohesion. As defined in the IEEE Glossary, cohesion is "the manner and degree to which the tasks performed by a single software module are related to one another. Types include coincidental, communicational, functional, logical, sequential, and temporal." [IEEE90] These types of cohesion are described in Table 3 below, with the wording taken primarily from the IEEE definitions, with some modifications, listed in order of decreasing desirability.

functional: the tasks performed by a software module all contribute to the performance of a single function.
sequential: the output of one task performed by a software module serves as input to another task performed by the module, in a one to one type of relationship.
communicational: the tasks performed by a software module use input data from, or produce output data, for other tasks within the module, in a potentially many to many type of relationship.
temporal: the tasks performed by a software module are all required at a particular phase of program execution or point in time.
logical (or procedural): the tasks performed by a software module perform logically similar functions, on different types of input data.

coincidental: the tasks performed by a software module have no functional relationship to one another.

Table 3. Types of Cohesion (After: [IEEE90])

Different modules will exhibit different kinds of cohesion. Ideally all modules should always have functional cohesion, which would result if modular decomposition was done on the basis of hiding design decisions as encouraged by Parnas. Reference to his work on the topic of modular decomposition has already been introduced above. While ideal, this is not always possible. Often other types of cohesion will be encountered in large software projects. Some examples of where each type of cohesion might be found are given in the table above.

4. Coupling

The next software design aspect to be covered is coupling. The IEEE Glossary defines it as "the manner and degree of interdependence between software modules." [IEEE90] The various types of coupling are listed in Table 4 below, in order of decreasing desirability. The coupling properties of a system indicate how modules communicate with one another and how strictly this interfacing is defined and enforced. The descriptions of the various types of coupling in the table below were drawn from the "Separation Kernel Protection Profile" [SKP04]. These definitions were chosen in favor of the IEEE Glossary definitions due to the greater degree of clarity and precision that they provide.

call: two modules are call coupled if they communicate strictly through the use of their documented function calls; examples of call coupling are data, stamp, and control, which are defined below.
data: two modules are data coupled if they communicate strictly through the use of call parameters that represent single data items.
stamp: two modules are stamp coupled if they communicate through the use of call parameters that comprise multiple fields or that have meaningful internal structures.

control: two modules are control coupled if one passes information that is intended to influence the internal logic of the other.
common: two modules are common coupled if they share a common data area or a common system resource. Global variables indicate that modules using those global variables are common coupled.
content: two modules are content coupled if one can make direct reference to the internals of the other (e.g. modifying code of, or referencing labels internal to, the other module). The result is that some or all of the content of one module are effectively included in the other. Content coupling can be thought of as using unadvertised module interfaces; this is in contrast to call coupling, which uses only advertised module interfaces.

Table 4. Types of Coupling (After: [SKP04])

5. Complexity

The final aspect left to define is complexity. One of the goals of security engineering should be to minimize the complexity of both the design and implementation of a system. Not only does this make the system easier to correctly implement, it results in a more maintainable code base. A programmer assigned to maintenance of the code should not be required to learn the inner workings of a complex design but should instead be able to quickly ramp up their understanding of the code with little up front investment. But specifying how complexity can be minimized is often done in terms of the four design aspects previously defined. A definition, adapted from the IEEE glossary, states that complexity is "a measure of how difficult software is to understand, and thus to analyze, test, and maintain. Reducing complexity is the ultimate goal for using modular decomposition, layering and minimization. Controlling coupling and cohesion contributes significantly to this goal." [SKP04]

As four of these five aspects – modularity, layering, coupling and cohesion – will be discussed separately in the context of OpenBSD below, the complexity section will simply focus on those measures of complexity related to minimization. These measures include degree of adherence to coding standards, existence of unused or legacy code and adherence to a high-level software architecture description.

C. OPENBSD ANALYSIS

1. Modularity Analysis

The first aspect that will be examined is the extent to which the OpenBSD system has been divided into distinct modules. There are two different ways in which modularity should be considered. It is crucial to first give a cursory look at how the system overall is decomposed into modules and how this impacts the initialization code. With that understanding, the degree to which the initialization sequence itself is divided into modules can be examined.

The OpenBSD operating system defines a number of different modules which serve to give the operating system structure and organization. Yet the overall monolithic design of the kernel still falls under Tannenbaum and Woodhull's "Big Mess" design category [TAN97]. While not separated into distinct programs, the OpenBSD kernel does make use of modular design to structure its subsystems, such as the Network File System (NFS) module within the virtual file system subsystem. This division of the system into a number of subsystems simplifies the initialization code to a large extent. Therefore each subsystem defines a module that handles initialization of that subsystem. Rather than having to directly manipulate the data structures internal to each subsystem, the main initialization routine simply makes a call to each subsystem's initialization module. This will be discussed further in the section on coupling below. The modular decomposition observed in the rest of the system has the effect of simplifying the implementation of the initialization routine. Modularity allows the startup code to make clean function calls that address each subsystem. An alternate architecture for defining modules could change how the modules interface with the initialization routine. Exploration of alternate architectures is outside of the scope of the discussion here.

In general, the principles of data hiding and encapsulation are practiced as they should be, hand in hand with modularity. Most modules encapsulate data that is then hidden from other modules within the system. Yet these principles are not strictly followed, as there are clearly a number of global variables used by the system. Header files which are included in nearly every source code file, such as */sys/systm.h* and */sys/param.h*, are clear examples of where these principles have been relaxed. Often the decision to do so is performance related. The very role of an operating system makes it particularly sensitive to performance penalties. Thus while the principles of data hiding and encapsulation are clearly in effect, they have not been strictly enforced within the OpenBSD code base.

Next our attention turns to how the tasks that must be performed during initialization have been organized into modules. In OpenBSD there are basically two modules that together define the kernel initialization sequence. First, a processor-specific piece of assembly code must run, found in */arch/<archname>/<archname>/locore.s*. This assembly code sequence will be referred to as simply *locore.s* in the discussion here. This is followed by the processor independent code found within *main()* in */kern/init_main.c*. As this is the only main() function found in the kernel code, all further references to *main()* can be assumed to refer to the function defined in */kern/init_main.c.* Clearly an effort has been made to write as much of the initialization routine as possible in a high-level language, the result of which has been this division into two modules.

Control is transferred from the boot loader *boot* to the *start* label within *locore.s*. Approximately 360 lines of assembly make up this module on an x86 OpenBSD system, and is executed before *main()* is called. The locore.s module establishes the most basic memory support such as creating the GDT, LDT, GDT memory segments and LDT memory segments, as well as initializing the IDT used by the system. While these are tasks that must be done using assembly language, it should also be possible to encapsulate this assembly code into a C language function using inline assembly. While this would condense the initialization sequence down to essentially one module, it begs the question: what advantage would this redesign have? Easier traceability from boot program to kernel entry point would perhaps be one argument, but as the current design is not overly complex (see discussion of Linux initialization for comparison), this is probably not sufficient cause to make such a change.

If reduction to a single module is the goal, an alternate idea would be to push the functionality of this assembly prelude into the boot loader. This would not be possible as the code in *locore.s* stands now. Currently the assembly instructions in *locore.s* make calls to C language functions, which access kernel data structures and call other kernel functions. As part of the boot loader, access to these data structures would no longer be possible.

A novel idea would be to create a new category of program, an advanced boot loader, that could establish much of the processor-specific memory state, such as the IDT. It would not be responsible for set up of OS-specific data structures. It could then transfer control to the kernel via a well-defined interface, allowing the kernel to build on what has already been established. It would free the kernel from much of the lower level set up that is still being done inside of operating systems, despite the sophistication of boot loader programs. And such an approach would have the advantage that, if done correctly, such a piece of code should be portable between operating systems. This has particularly important implications for high assurance systems, as such a program could be evaluated once and continually reused.

While this would seem to provide a great deal of value to a high assurance development project, the need for such a redesign of the OpenBSD initialization sequence hardly seems necessary. As the amount of assembly that initially runs is fairly small, it is appropriate to have it comprise one module while the *main()* function comprises the other piece of the initialization picture.

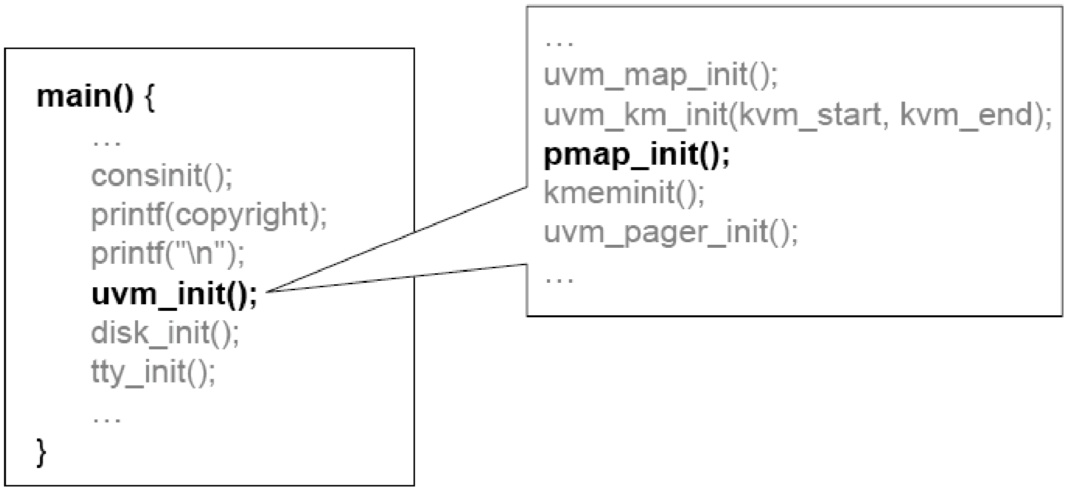
From examining the degree of modularity in the OpenBSD system there are two conclusions that can be drawn. First, a large number of carefully designed modules in a system contributes to the overall clarity of initialization. It can decrease the number of lines of code needed in the high level initialization routine, in this case *main()*, as each module can provide initialization for a single subsystem. Secondly, it is appropriate for a medium robustness operating system to have a couple of modules that handle initialization tasks. Any further subdivision would complicate interdependencies that are a result of having a monolithic kernel. Yet subdivision, such as creating a separate program between the boot loader and kernel, may be an interesting concept to explore when a higher level of assurance is required.

2. Layering Analysis

The next aspect of the OpenBSD initialization routine to explore is the degree to which layering is utilized. As with modularity, we must look at the layering of the rest of the system as well as that of the initialization code. Layering is used primarily in the OpenBSD system in order to separate hardware dependent functions from hardware independent functions. A clear effort was made to make the original operating system as hardware independent as possible, in 4.4BSD only about 7% of the code was hardware dependent [MCK96]. A useful demonstration of this kind of layering can be found in the way that OpenBSD implements memory management.

The memory management subsystem consists of two main modules layered on top of each other, each of which is responsible for one type of mapping. The higher layer uvm module deals with memory in terms of vm_maps and vm_pages. It maps these virtual address spaces to specific files or objects. The example given in a code comment is that this layer would know to "map virtual addresses 0x1000 to 0x22000 read-only to the file /bin/ls starting at offset zero." At this higher level, there is no thought given to how the virtual vm_pages are physically mapped in memory. This mapping of virtual pages to their physical location is handled by the pmap module. For example, when a page fault occurs the fault routine will establish which virtual memory page is needed and the upper layer will request that the lower layer, the pmap module, create a mapping for it. Thus the uvm layer depends on the pmap layer to provide hardware-specific services.

These two layers are properly oriented with respect to the runtime code, following the rule that a "layer depends only on layers below it in the hierarchy for services, and provides its services only to the layers above it." [SKP04] Yet this fundamental notion of layering is not preserved with respect to initialization. The primary initialization function, *main()*, does not initialize the pmap module directly but instead does so through the uvm module. To initialize the uvm module, the function *uvm_init()* is called from *main()*. Within *uvm_init()* a call is made to *pmap_init()* which then handles initialization of the pmap module. Figure 11 illustrates the way in which the function calls are structured within the source code. Thus the lower layer actually depends on the higher layer to provide initialization services. While perhaps the intent of this design was to provide data hiding, thus hiding initialization of the pmap module from the main() function, it is not appropriate for a lower layer to have this type of upward dependency. Thus while the idea of layering appears within the OpenBSD system, a strict layering policy is not enforced throughout the system.



```
main() {
    …
    consinit();
    printf(copyright);
    printf("\n");
    uvm_init();
    disk_init();
    tty_init();
    …
}
```

```
…
uvm_map_init();
uvm_km_init(kvm_start, kvm_end);
pmap_init();
kmeminit();
uvm_pager_init();
…
```

Figure 11. Execution Flow of *uvm* and *pmap* Initialization

As concerns the two initialization modules defined in the previous section, the initial assembly code and *main()*, there is no layering between the two modules. They effectively operate within the same layer. Instead the two modules are simply arranged sequentially.

In general, most of the functions called from within *main()* simply establish a handful of data structures. For example, the function *soinit()*, which initializes sockets, has a one line implementation calling the memory management subsystem to establish a "pool" of sockets[4]. Thus the issue of layering is simply avoided for the most part by functions called from within *main()*. The majority of functions called within *main()* are related to a single module and initialize data structures for that module only, relying on the memory management subsystem to be operational. Yet when layering is observed, as in *uvm_init()* discussed above, it has been done in such a way as to violate the main tenet of layering. Therefore the initialization sequence of OpenBSD does not properly implement the principle of layering.

[4] A pool is type of object defined by the memory management subsystem which provides an efficient way for part of the kernel to reserve kernel memory for its own usage without having to implement its own memory allocator.

3. Cohesion Analysis

In discussing cohesion within OpenBSD, the type of cohesion visible depends heavily on the point of view being taken. Depending on the perspective one adopts, various types of cohesion can be seen in the design and implementation of the initialization sequence.

Both modules that make up the initialization sequence will by their very nature exhibit temporal cohesion. Generally such modules are not desirable. This is because a division of labor based on when tasks are supposed to happen is a tricky proposition in an operating system. This is generally a recipe for creating race conditions and potential deadlocks. Since a temporally cohesive module is not logically confined to a specific purpose, it is often difficult to understand. This makes bug-free modifications quite difficult.

Despite this undesirability, the initialization sequence is one of the cases where a temporally cohesive module is necessary and appropriate. In order to get the system up and running there are a number of tasks that must be done, all of which are only related insomuch as they all must be done at startup time. Therefore both the initial assembly language module, *locore.s*, and *main()* exhibit temporal cohesion.

Within the *main()* module, the initialization modules specific to various subsystems are invoked. These modules, responsible for the initialization of data structures relative to a specific subsystem, clearly exhibit functional cohesion. The single purpose of such modules is to prepare their respective subsystem for operation. This tends to primarily involve establishment of data structures for future use. The singular nature of this operation is a hallmark of functional cohesion.

In a monolithic kernel the degree to which all aspects of the system are intertwined dictates initialization cannot proceed in a random order. There exist sequential constraints on when specific initialization tasks happen. Careful design must ensure that no incorrect assumptions are made about the state of the system as a whole. For example, no calls to *malloc()* can be made until the memory management subsystem is properly initialized. It is clear from examining the OpenBSD code that creating such a design for a monolithic kernel is neither natural nor easy. There are a number of examples within the *main()* function of OpenBSD's initialization sequence where timing and sequencing are interrelated and must be done carefully. For example, line 209 of /kern/init_main.c reads "disk_init(); /* must come before autoconfiguration */." Looking down to line 214, a similar comment can be found, "Initialize mbuf's. Do this now because we might attempt to allocate mbufs or mbuf clusters during autoconfiguration." Thus the *main()* function actually exhibits sequential cohesion within temporal cohesion. While these functions do not provide direct input and output to one another, as the definition of temporal cohesion would require, the entire state of the system can be considered to be the input and output. Thus the output of *disk_init()* can be considered to be the state of the system that it creates, which is required as input for autoconfiguration to be able to take place.

While sequential cohesion is indicated by a one-to-one relationship between functions that exchange input and output within a module, communicational cohesion can be identified by the one-to-many nature of functions within such a module. An example of this type of cohesion would be a module that used a customer ID number to locate the customer's address in a contact database and their loan balance from another database and presented both items. As the sequence of data retrieval does not matter, this example is appropriately labeled communicational cohesion and not sequential cohesion. Within the OpenBSD initialization sequence there are no clear cases of communicational cohesion. Due to the nature of initialization, sequencing tends to be of importance.

Logical cohesion and coincidental cohesion are not observable in the initialization code as well. While aspects of the initialization code may appear to demonstrate only very loose cohesion, in fact temporal cohesion still binds the code together. Somewhat unrelated tasks seem to have been thrown together into functions like *cpu_configure()*, which includes some code that is not CPU-specific. But in fact these tasks are bound together at the very least by the fact that they must all take place during initialization. So, in truth, any part of the initialization that seems to have just logical or coincidental cohesion actually has the stronger bond of temporal cohesion simply due to the timing of when the code will be executed.

While the temporal cohesion that predominantly defines the initialization modules is not the most desirable, it is simply unavoidable. This undesirable form of cohesion is at least slightly offset by the functional cohesion that is used to a large extent by the smaller modules which make up the two initialization modules. While the temporal and sequential nature of initialization tasks cannot be changed or eliminated, it can be controlled by organizing the initialization modules to rely on functionally cohesive modules to the largest extent possible.

4. Coupling Analysis

There are two different areas where coupling should be considered. The first is the type of coupling that occurs within the initialization sequence in relation to other modules. The second is the way in which the kernel initialization sequence is coupled with the boot loader that runs just prior to the initialization code.

The type of coupling found within the OpenBSD initialization routine itself is exclusively call coupling. For example the assembly language that is initially run uses calls to C functions, not jumps or direct manipulation of memory locations, in order to handle a number of tasks during startup. Similarly, the transfer of control from this initial assembly code to the C language *main()* function is handled via a call. This compares quite favorably to the way in which the initial code run within the Linux kernel is coupled. As documented in the section on the Linux initialization sequence, the initial pieces of code that run are coupled quite awkwardly, for example using jumps to direct memory addresses to transfer control.

In OpenBSD, initialization in architected in such a way that each module defines its own initialization function within the module itself. The *main()* function then makes a call to each of the various initialization functions provided by these modules. For example, the function *tty_init()* is responsible for initialization of the tty module which manages terminals. The function creates a tail queue for storing all the tty's defined on a system, then sets the variable *tty_count*, which is global with respect to the tty module, to zero. This function is part of the tty module and accesses variables that are internal to the tty module. Thus the fact that *main()* calls *tty_init()* to initialize the tty module indicates that those two modules are strictly call coupled, as no sharing of global variables is done. More specifically, they are data coupled, since they strictly use call parameters, namely empty call parameters. The function *main()* does not directly change the variable *tty_count* itself but instead does so through a well-defined interface, the *tty_init()* function.

There are also examples of stamp coupling, another type of call coupling, within the initialization routine. One line found within *main()* is *uvm_init_limits(p);*. The *uvm_init_limits()* function takes a *proc* structure as an argument. The *proc* structure is an example of a call parameter that has multiple fields and a meaningful internal structure, a clear example of stamp coupling. While stamp coupling can be found in a number of places within the main() function, there are not any examples of control coupling that can be found. Rather the initialization sequence is limited to using data and stamp coupling, two types of call coupling, internally.

The way in which the kernel image is coupled with the boot loader should also be considered. Since both are compiled as standalone executable ELF format files, it may seem that they are call coupled, but in fact there is a closer relationship. These two programs are actually content coupled, as will be shown. This content coupling is a consequence of the kernel needing to utilize the same GDT as was used by the boot program. Lines 406-7 of */arch/i386/i386/locore.s* read, “Oops, the gdt is in the carcass of the boot program so clearing the rest of memory is still not possible.” The rest of memory cannot be cleared and set to a known state at this point, since the GDT in the “carcass” of the boot program is still in use.

This type of coupling is not all that unusual to find at this level in the system. For example the boot loader for Linux is supposed to write a number of values into the kernel image to communicate such information as the type of boot loader used and a pointer to the kernel command line location (see */Documentation/i386/boot.txt*).

While undesirable, content coupling is hard to eliminate entirely from the BIOS-to-kernel bootstrap process, as there are a number of special limitations and requirements as the system boots. Nevertheless, this kind of undesirable coupling underscores the need for such interfaces to be well-defined and documented. As it stands currently, understanding of the interface must be drawn exclusively from assembly code that often uses direct address location references. As mentioned previously, a “Multiboot” standard has been proposed by the Free Software Foundation. Wide acceptance of such a standard would greatly help the clarity of the entire bootstrap process as a whole. In particular it would ensure a more structured interface between the boot loader and initialization code.

In summary, coupling within the OpenBSD kernel, and in particular within the initialization function *main()*, is managed well and limited to call coupling. The biggest source of less desirable coupling is the relationship between the boot loader and the kernel. These observations will be revisited when recommendations for the design and implementation of the initialization sequence are proposed in Chapter VII.

5. Complexity Analysis

As complexity is often defined in terms of modular decomposition, layering, coupling and cohesion, which are discussed above, this section will focus on measures of complexity not yet addressed, namely those closely related to minimization. These measures include degree of adherence to coding standards, existence of unused code and adherence to a high-level software architecture description [SKP04]. Each of these three complexity indicators will be discussed in the section that follows.

The OpenBSD project actively maintains a man page devoted to specifying a coding standard. At the time of this writing the CVS (Concurrent Versions System) logs indicated that this style guide had most recently been updated about six months ago. The guide focuses on the formatting of both code within the kernel and code for programs to be included in the OpenBSD distribution. While no rigorous checking was done, all of the code encountered by this author appeared to adhere quite well to this standard. In particular, the initialization code found within the *main()* function adheres closely to this coding standard.

Much harder to standardize is the usage of functions based on context. Specifically, when multiple functions perform a similar or identical task, lacking usage guidelines, complexities in the code can develop. For example, in the OpenBSD kernel code, the same assembly instructions or functions are not always used to enable and disable interrupts. In assembly files, the common convention appears to be to use the *cli* instruction to disable interrupts and the corresponding *sti* function to enable interrupts. These instructions respectively clear and set the IF flag in the eflags register [INT03-2].

Yet one case was encountered where a *cli* instruction appeared to have no corresponding *sti* instruction (see */arch/i386/i386/microtime.s*, line 57). Any programmer with a fairly good grasp of assembly would quickly recognize that the *popfl* instruction found in the code has the effect of enabling interrupts. It should be noted that a comment on that instruction line indicates as much. The instruction writes the entire eflags register with the value that was on top of the stack, thus writing over the interrupts flag. Similarly there are cases where the function *disable_intr()*, which typically has a corresponding *enable_intr()* function, is instead followed by the call *write_eflags(save_eflags)* (see /arch/i386/isa/npx.c, line 303). The effect is the same, yet this lack of consistency does make the code more difficult to read. These examples clearly do not pose huge barriers to understanding the code, but they are indicative of the types of issues that can arise in a large software project. While clearly not of significant consequence for the OpenBSD system, these types of inconsistencies should certainly be avoided in a high assurance system.

Another consideration when looking at complexity is the extent to which unused code exists in the code base. Any unused code should be removed from a software project once it is identified as unused. This serves to reduce the code base that must be maintained and any confusion that may arise out of its existence. If there is concern that the code may need to be reintegrated in at a future date, this should not prevent the code from being removed. With proper configuration management in place, the task of retrieving an earlier version of a source code file should be trivial. A search through the OpenBSD source code for the word “unused” turned up a number of lines of code that are no longer used by the kernel and marked as such. For the most part these consisted of *#define* statements which define a name for a constant that is never referred to elsewhere in the code (see */dev/pci/pciide_natsemi_reg.h*, lines 80 and 85). While essentially harmless, this indicates there is not a ruthless push to slice all unused code out of the kernel. Care is taken by OpenBSD developers to ensure that unused code is not compiled into the binary (see */dev/raidframe/rf_aselect.c*). The *#if* construct works effectively in this case, if there is a compelling reason to keep the unused code. For example, it may serve as a template or guide for how to write code to interface with a particular module.

As long as such code is documented as unused and then isolated, as is the case here, it does not pose a problem.

Another key piece of the software design process which can aid in reducing complexity is the development of, and adherence to, a high-level software architecture design description. In the case of OpenBSD, the developers have stated quite publicly that there is no “master plan” or design document that they are working towards [MIS01-1]. Rather the project has always been about developing a secure operating system that meets the needs of the developers themselves. Thus features are added incrementally as the need for them develops, a very evolutionary process. As any type of “roadmap” has the potential to create bad publicity if not fully realized, such lists of promises have been avoided [MIS01-2]. In general, most open source development projects do not spend time creating documentation of a high level design in the way that would need to be done for high assurance design. Rather they rely on a much more evolutionary process.

The above discussion, while not rigorous, should have provided a reader with a sense of the issues that can give rise to complexity in a large-scale software design project. In order to minimize complexity, an awareness of these issues must be present. While there are areas of improvement possible for the OpenBSD system, an examination of the code conveys the sense that the developers are acutely aware of many of these pitfalls.

D. SUMMARY

In the preceding sections the degree to which OpenBSD adheres to widely recognized security engineering principles has been examined. While no attempt has been made to quantify this adherence, this discussion should have given the reader a good sense of the degree to which this open source operating system utilizes design principles essential to assuring the security of a system. As discussed earlier, this examination serves a dual purpose. With an understanding of how well a system has been engineered, one can more easily gauge the likelihood of an existing vulnerability. But code that has been carefully engineered is also less susceptible to unintentional errors, as well as an intentional insertion of malicious code. As the discussion of threats determined that both should be of concern to the initialization sequence, an examination of how security engineering principles can be applied as a countermeasure was called for. While clearly such principles have been applied to OpenBSD to some extent, weaknesses were also revealed. Based on the threats discussed in Chapter V and the weaknesses in security engineering explored here, it is now possible to draw up concrete recommendations for how to design the initialization sequence of an operating system for maximum robustness.

VII. RECOMMENDATIONS

A. INTRODUCTION

Best practices for architecting and implementing an initialization sequence can now be explored, having finished discussing the security engineering aspects of initialization and relevant threats. In general, these recommendations are high-level, having to deal with design and architecture. Yet, when possible, specific details are given on how each recommendation could be realized in an actual production operating system. The intent is to avoid overly general or obvious recommendations such as "use good software engineering practices" and instead focus on design aspects that are unique or interesting to the initialization routine. It is assumed that the initialization sequence will be subjected to the same lifecycle requirements as the rest of the system.

While these recommendations have arisen out of an examination of OpenBSD and Linux, these recommendations are not targeted at those systems in particular. They are intended to be of use to developers and system architects across the spectrum, providing guidance on how to build a robust initialization routine. The discussion that accompanies each recommendation also indicates the extent to which it is applicable to a medium vs. high robustness system. While all recommendations will be applicable to a high assurance system, some will be equally applicable to a medium robustness system, whereas others simply do not make sense when targeting a lower robustness level.

Each recommendation is listed in Table 5, along with a shorthand identifier. Every recommendation stems from one of the identified threats in Chapter V or one of the security engineering principles in Chapter VI, which relate back to the POOR_DESIGN and POOR_IMPLEMENTATION threats. Some recommendations identify good practices that are already in place, but perhaps not implemented correctly, such as R.1. Others are inspired by significant research that has already been done in a related area, such as R.3. Still others draw from work which is still very much in the development stages, such as R.4. Each of these recommendations will be discussed in detail in Section C.

Recommendation Identifier	**Description**
R.1	Prevent access to the initialization code once the runtime code is executing to reduce potential exploitation of vulnerabilities in initialization code.
R.2	Strict coding standards should be enforced to enhance clarity and understandability.
R.3	Integrity checking of each stage of the boot process should be done by the code that loads the subsequent stage to ensure that the code at each stage has not been altered.
R.4	A widely accepted standard for the interface between the boot loader and operating system kernel needs to be developed to provide additional transparency and promote software reuse.
R.5	Layering relationships must be preserved within the initialization routine to avoid circular dependencies.
R.6	The boot loader and operating system should be strictly call coupled to reduce complexity.
R.7	All interactions of the kernel with other components during initialization must be logged to provide sufficient audit data.
R.8	Create an initial secure state specification and map the implementation to the specification in order to verify the initial state.

Table 5. Initialization Recommendations

The correspondence between the threats identified in Chapter V and the recommendations proposed here is outlined in Table 6 below. While most threats can be countered by one of these recommendations, some require other countermeasures unrelated to the initialization software. These are noted as “Procedural countermeasures” in Table 6 and are discussed in the next section.

Relevant Threat	**Recommendation**
POOR_DESIGN	R.4, R.5, R.6
POOR_IMPLEMENTATION	R.1, R.2, R.4, R.5, R.6
POOR_TEST	R.4
UNIDENTIFIED_ACTIONS	R.7
UNKNOWN_STATE	R.8
ALTERED_DELIVERY	R.3
INSECURE_STATE	R.8
ADMIN_ERROR	Procedural countermeasures
ADMIN_ROGUE	Procedural countermeasures
AUDIT_COMPROMISE	Procedural countermeasures

Table 6. Mapping of Threats to Countermeasures

B. PROCEDURAL COUNTERMEASURES TO THREATS

There are a number of threats that were determined to be applicable to the initialization sequence in Chapter V, yet are not directly addressed by any of the initialization recommendations discussed below. These threats require procedural and administrative countermeasures rather than technical countermeasures. In the interest of completeness, it is important to demonstrate how all of the threats discussed in Chapter V could and should be addressed. Therefore this section is devoted to a discussion of countermeasures that themselves are unrelated to the initialization routine, but yet counter a threat that does concern the initialization routine. Many of these are countermeasures unrelated to engineering that could not be implemented or enforced using technical means.

1. Countermeasures to ADMIN_ERROR

The threat ADMIN_ERROR was determined to relate to the initialization sequence, since OpenBSD allows an administrator to specify kernel configuration options at initialization time. Input of an improper configuration could give a malicious device unintended access to the system. Yet this threat does not strictly call for engineering measures to be taken within the initialization sequence. Certainly, all input should be carefully checked by the initialization code to ensure that it is of a valid format and is consistent. Yet it would not be possible to include a software mechanism within the initialization sequence that could determine if a given configuration was "secure" or not. The very definition of "secure" could vary too widely, based on the needs of the site, to be correctly automated. Thus additional measures are necessary. One approach would be to reduce or remove altogether the ability to input this type of configuration data. Such a solution may be appropriate for a high assurance system that does not require frequent hardware changes or reconfiguration. Yet limiting this functionality greatly impacts usability. A less restrictive countermeasure would be to train all administrators to be able to properly configure a system for security. This type of training, combined with some automated validity checking of all inputs, should provide sufficient protection against the ADMIN_ERROR threat, as it relates to initialization.

2. Countermeasures to ADMIN_ROGUE

The threat ADMIN_ROGUE is related to the above threat, but is concerned with intentional misconfiguration or damage. An administrator typically has full control over a system, including configuration during the initialization process. It is essentially impossible to restrict administrator actions with automated software since such a mechanism would have a hard time distinguishing rogue behavior from normal administrator behavior. For example, deletion of a file could cause the system to become unstable or it could be part of normal housekeeping tasks. Therefore additional countermeasures must be in place.

A wise practice is to perform some kind of background check for any employees who will be involved in sensitive tasks. While past performance is not always indicative of future behavior, it certainly provides a way to filter out high-risk employees. The very notion of a root user who has complete control over the entire system is one to be avoided. Thus it would be prudent to practice separation of duties, essentially applying the principle of least privilege to employee roles by limiting the amount of control any single administrator has over the system. For example, an administrator responsible for adding and deleting user accounts only needs access to that functionality and should not be allowed to change the networking configuration of the system.

Finally, administrators should also be required to use an identification and authentication mechanism to access the system which would tie in to an audit capability that could not be altered. Knowing that their actions could be traced back would serve as a strong countermeasure as well. Taken together, these actions provide a number of ways that the initialization sequence could be protected against this threat.

3. Countermeasures to AUDIT_COMPROMISE

To protect the audit log generated at initialization, typically called the *dmesg* on a UNIX system, the initialization code must employ the same kinds of protection as the rest of the audit system to counter this threat. Ensuring that audit events are properly captured at initialization simply requires good design and implementation. But to ensure the audit record cannot be destroyed or altered requires some intelligent countermeasures. One technique is to save all audit logs to write-once media to ensure they cannot be altered. Another method would be to send audit data to another machine, which could, in turn, store it offline. Along these same lines, mechanisms should be in place to handle the situation where audit logs become full. In a medium robustness system, the design may call for old audit logs to be deleted when the audit capacity becomes full, or new incoming audit data may simply not be captured. But for a high assurance system, these two alternatives are generally insufficient. A solution that avoids loss of audit data would be to require that a machine simply shut down once the audit capacity becomes full until capacity can be made for the additional incoming log data. It is a sufficient countermeasure to simply ensure that the audit data generated by the initialization sequence receives the same type of protections as that generated by the other subsystems.

C. RECOMMENDATIONS

Presented below are the recommendations for designing and implementing the initialization routine of an operating system, based on the analysis of the preceding chapters. Each recommendation is presented with an overview of its purpose and implications. A more in-depth explanation of how the recommendation can be implemented is found in the "Discussion" section of each recommendation.

1. Recommendation 1

R.1 Prevent access to the initialization code once the runtime code is executing to reduce potential exploitation of vulnerabilities in initialization code.

The first recommendation is to limit the ability of the initialization code to be executed. In other words, the initialization code should be allowed to run once and then be removed from memory. The purpose of such a recommendation is tied to the principle of minimization. Essentially it reduces the amount of code present in memory during runtime, reducing the chance that reuse of initialization code could lead to execution of undesirable sequences. For example, even though attackers might find a flaw that allowed them to control the point of execution, the execution point could be limited to the bounds of the kernel code. The ability to arbitrarily move execution into initialization code is a concern due to the lack of scrutiny typically applied to this code and the *ad hoc* way in which initialization code is structured and written. Returning execution to the initialization sequence could lead to exploitation of an existing vulnerability and fulfillment of additional attacker goals that otherwise would not be possible. Therefore to reduce the impact of possible flaws within the initialization sequence, it is a prudent practice to remove this code following its execution.

a. Discussion

In terms of actual implementation, there are multiple ways to ensure initialization code is not run again. As discussed in Chapter IV, the Linux operating system uses a feature of *gcc* that on first glance appears to prevent access to the initialization code once it has run. The *gcc* compiler provides the programmer with the ability to specify various "attributes" of code. Using the "attribute" tag, a programmer can specify which section of the binary a specific part of code should be linked into. Linux defines the *__init* macro, which ensures that any function tagged with this macro is compiled into the *.text.init* section of the ELF binary. Once all the initialization functions have completed, the memory that the *.text.init* section occupies, as well as memory occupied by other initialization related sections, is freed by the function *free_initmem()*. While the pages of memory are freed, they are not actually overwritten or zeroed out at this point. Linux delays zeroing of memory until allocation, and in fact it is possible for kernel space code to allocate pages without having them zeroed, via the *__get_free_page()* function. Thus future access to the initialization code is not prevented until the pages containing initialization code are reallocated and overwritten.

It is likely that the freeing of these memory pages in Linux was initially implemented for performance reasons. On a very small device, the ability to reclaim the memory occupied by initialization code could be quite useful. Yet while these attribute tags appear to also serve a security function, they in fact do not. OpenBSD does not appear to segregate the memory occupied by initialization code in any way from runtime code.

A better approach to not allowing the initialization sequence to execute again would be to use hardware support. Using the segmentation support provided by the Intel chip, the initialization code could occupy a separate segment, access to which could be denied after initialization by removal of the descriptor for this segment from the global descriptor table. Alternately, new hardware privilege levels could be added to support the initialization code, separate from the privilege level of the kernel. The feasibility of such a design is an area of future work, and thus will not be discussed at length here. Implementing either example solution outlined here would ultimately provide significant protection against the POOR_IMPLEMENTATION threat, either by a malicious or careless developer.

2. Recommendation 2

R.2 Strict coding standards should be enforced to enhance clarity and understandability.

The next recommendation is to develop and enforce strict coding standards, which would include detailed instructions on acceptable coding practice. The purpose of such standards would be to reduce complexity and eliminate confusion about the purpose of various sections of code. As initialization code is typically fairly complex to begin with, introduction of any unnecessary complexity must be limited. The creation of coding standards would reduce the threat of either unintentional or intentional implementation errors, POOR_IMPLEMENTATION.

a. Discussion

In particular, the use of hard-coded jumps to specific physical locations should not be allowed unless absolutely necessary. Similarly, usage of duplicate function names should be avoided. In the Linux kernel initialization code, both of these poor coding practices collide. The result is two functions with identical names that yet do not confront any scoping issue, since the functions are not reached via a call to a function or label name but instead are reached by hard-coded jumps. While the hard-coded jumps alleviate the problem of conflicting names, the result of combining jumps to physical addresses and conflicting names is confusing code, the correctness of which is difficult to ascertain. Additionally, many levels of nested macros, as seen in the discussion of the Linux kernel in Chapter VI, should be avoided.

The need for such standards is clear, but the question of how to go about enforcing such standards is more tricky. The best solutions will be automated in order to support consistent management of a large code base that has perhaps been touched by many developer hands. Unfortunately, if standards are to be customized to the programming language, target platform and project, any tool developed to do automated standards enforcement must necessarily be customizable as well, in order to be reusable. While exploration of the existing tools that fit into this category is outside the scope of this thesis, such tools clearly play an important role in the development process of the initialization code, as well as the system as a whole.

3. Recommendation 3

R.3 Integrity checking of each stage of the boot process should be done by the code that loads the subsequent stage to ensure that the code at each stage has not been altered.

In order to counter the threat of ALTERED_DELIVERY, integrity checking of each stage in the boot process is required. Defending against this threat is a two-part process. Initially, procedures need to be in place to ensure that the code delivered to the operational site from the development site is the correct and intended code. Once at the operational site, the operating system integrity must be verified on an on-going basis to ensure that the system has not been altered or patched by an adversary since delivery. If on-site, on-going verification of an operating system is not done, it is possible that at some point after delivery the running system could be altered and is no longer the same system that the developer intended to deliver. To ensure that no such tampering has taken place, integrity checks should be in place.

a. Discussion

As outlined in the Chapter II overview of the boot procedures, an architecture for ensuring the integrity of each stage of execution while booting an operating system has been developed by Arbaugh [ARB99]. It provides a framework for using cryptographic signatures to ensure the integrity of the software installed on a system. On some level this is mostly a concern of the BIOS and boot loader, and is outside the scope of kernel initialization. Yet, since the boot loader and initialization sequence have a very close relationship, it is worth considering here. Additionally, such a verification mechanism could be used to verify the *init* program running in user space. While not discussed at length in this thesis, the integrity of the *init* program certainly has an effect on the assurance of a system. Thus this type of verification mechanism, while involving aspects of the system beyond just the kernel initialization sequence, is such a key component of assured operation of a system that it merits a recommendation.

4. Recommendation 4

R.4 A widely accepted standard for the interface between the boot loader and operating system kernel needs to be developed to provide additional transparency and promote software reuse.

Currently there are no widely-recognized standards for how a boot loader should interface with the operating system it is booting. The benefits of such a standard would be twofold. First, simply having such a standard documented would be useful in and of itself, bringing a degree of clarity that does not currently exist. This clarity would make validation of adherence to the standard easier, as such a standard would encourage the development of standard tools to use for development and testing. A second benefit would be the ability to reuse previously developed boot loaders. These benefits would directly counter the threats of POOR_DESIGN, POOR_IMPLEMENTATION and POOR_TEST. The example section below describes how such a standard could be developed and the benefits that would be derived.

a. Discussion

In terms of concrete implementations of this recommendation, the existence of the Multiboot standard supported by the Free Software Foundation is a strong first step. As has already been noted in previous chapters, this is an effort to standardize the way in which a boot loader should interface with an operating system. The key to this recommendation is that it will require work on the political end of the technology spectrum, beyond the technical work required. To receive widespread acceptance, operating system developers would have to be convinced that such a standard was useful even though it may not appear to be in their best interest. While development and adoption of standards is not an easy process, since widely accepted standards typically lead to a more open marketplace, such standards often end up being in the best interest of those who initially resist them.

With a widely-accepted standard in place, a single trusted boot loader would only have to be developed and certified once and could then be used widely by high assurance systems. This would reduce the development effort required to field a new high assurance system. In order to create a boot loader that could be widely used by high assurance systems, not only would it have to adhere to a boot interface standard but the boot loader itself could not be too complex. To be certifiable, it would be in the interest of the developers to minimize its functionality and thus complexity. In the open source world, it can be seen that current development of GRUB 2 is going in the opposite direction. New functionality is being added, such as a memory management capability and the ability to dynamically load modules when GRUB is run to extend the boot loader's capabilities [GRU04]. These types of efforts, while useful to some, do not serve security objectives.

In terms of drawbacks, there are certainly aspects of the initialization routine that are too closely tied to kernel operation to allow a program distinct from the kernel to perform them, such as initialization of kernel data structures. Yet there are tasks that could be handled this way. The OpenBSD operating system provides a nice example of the kinds of tasks that can be delegated to a fairly sophisticated boot loader, such as switching into protected memory mode. But it must be recognized that there are trade-offs between having a standardized boot loader and a highly tailored boot loader. Any boot loader standard would have to be fairly generic in order to gain widespread acceptance, thus its functionality would likely not be very sophisticated. On the other hand, a boot loader built for a specific operating system could be highly tailored and handle many more tasks that traditionally reside within the kernel initialization, reducing kernel complexity. But the latter case completely bypasses the benefits of software reuse gained in the former scenario. This trade-off should be weighed carefully, particularly for a high assurance system, as the benefits of both sides are attractive when trying to engineer for security. While it may appear that it is simply a question of shifting the complexity around, if a boot interface standard could be developed that also moved a significant amount of complexity into the boot loader, the benefit would be in reusing this boot loader for many high assurance systems. To make building and utilizing such a boot standard more attractive than using a very tailored boot loader, perhaps a handful of boot loader-operating system interface standards, of varying levels of functionality, need to be proposed.

Despite these trade-offs, the development of a boot loader-to-operating system standard, and its widespread acceptance, is of distinct benefit to the development and implementation of the initialization routine of operating systems. With a well-defined standard to work from, the tasks that must be done by the initialization routine would be clear, and could even be minimized by a more sophisticated standard.

5. Recommendation 5

R.5 Layering relationships must be preserved within the initialization routine to avoid circular dependencies.

Layering relationships that exist in the runtime code should not be bypassed during initialization. The purpose of layering modules is to ensure controlled access between different parts of a system and to eliminate circular dependencies between modules. Proper layering ensures that a layer depends only on layers below it for services, and provides its services only to the layers above it. This ensures that changes to higher layers do not affect lower layers, as there are no circular dependencies. Enforcement of layering relationships serves as a countermeasure to POOR_DESIGN AND POOR_IMPLEMENTATION, as changes made to higher layer modules will not result in unintended consequences for lower layers. It is crucial that such relationships are preserved within the context of initialization. Otherwise, there is the potential that changing how a higher layer initializes will result in a lower layer being improperly initialized.

a. *Discussion*

A circular dependency scenario was observed within the OpenBSD memory management subsystem, as noted in Chapter VI. The lower layer *pmap* module actually depends on the higher layer virtual memory module to provide initialization services. Therefore changes to how the higher layer handles initialization could cause *pmap* to not be properly initialized.

To prevent these types of interdependencies, and the potential implementation errors that they can create during maintenance of a code base, strict layering should be enforced in the initialization code as well as in runtime code. How this can actually be enforced is not easy to answer. Use needs to be made of tools and techniques such as mapping the implementation back to the design and automated tools to trace the relationships between various aspects of the code. These can all contribute to the proper design and implementation of layering within the initialization routine, as well as within the operating system as a whole.

6. Recommendation 6

R.6 The boot loader and operating system should be strictly call coupled to reduce complexity.

This recommendation is intended to counter the threats POOR_IMPLEMENTATION and POOR_DESIGN by specifying a stricter clear cut interface, reducing complexity. As seen in the preceding chapters, the way in which the boot loader and operating system are coupled is often quite complex. Some of the examples discussed in previous chapters are reviewed below.

a. *Discussion*

This recommendation is tied to R.4 regarding a boot interface standard. It goes beyond that recommendation to explicitly state that any such interface should only provide for strict call coupling between the boot loader and operating system. Chapter VI discussed how the OpenBSD kernel uses the GDT found within the boot program's memory, rather than generating one of its own. Linux similarly is not strictly call coupled, as the boot loader is expected to fill in certain header values of the kernel image upon boot. Details on these values can be found in */Documentation/i386/boot.txt* and the source code file */arch/i386/boot/setup.S*. While these strange types of coupling are clearly essential to the way in which these operating systems boot, this is not inherent to the boot loader and operating system relationship. Instead the relationship can be clearly defined and linked only via well-defined interface, as the Multiboot standard demonstrates. There, the only communication is handled via a register value. In the Multiboot standard, when the operating system starts executing there exists a pointer to a structure in the ebx register. This structure contains any data that needs to be passed from the boot loader to the operating system, such as hard drive geometry or the boot loader name [OKU02]. The information contained in this structure is strictly advisory and may be used by the operating system as it sees fit.

7. Recommendation 7

R.7 All interactions of the kernel with other components during initialization must be logged to provide sufficient audit data.

To counter the threat UNIDENTIFIED_ACTIONS it is necessary that appropriate and relevant security-related events are recorded in the initialization audit log. Relevant events include the interactions of device drivers with devices as they are probed and initialized and any loading of additional kernel elements, such as kernel modules, if allowed. The most important information that must be conveyed by the initialization audit log is the types of interactions that the kernel is having with outside devices and software.

a. Discussion

Specifically, it is important to know is what types of devices have been loaded into the system, what device drivers are associated with them and what type of behavior they are exhibiting. Similarly, it is vital to know if any kernel modules are loaded at boot time, so their presence can be validated as expected or not. As long as the initialization sequence records these types of events, it can then rely on other aspects of the operating system to properly protect this audit data. As the kernel itself should be signed and verified, per R.3, the fact that the kernel creates a queue for storing the process list is not necessary to audit. As this is included in the kernel code, which has not been altered, it provides no new or interesting security relevant information. The key audit concern that developers must think about with respect to the initialization routine is the type of data to be recorded, and to ensure that any interactions of the kernel with either hardware devices or other software modules is recorded.

8. Recommendation 8

R.8 Create an initial secure state specification and map the implementation to the specification in order to verify the initial state.

In order to counter the threats UNKNOWN_STATE and INSECURE_STATE it is necessary that the initialization routine can be shown to reach an initial secure state. This recommendation is one way that assurance can be provided that the initial state reached is secure. But before delving into this recommendation, the relationship between these two threats should be clarified.

Both UNKNOWN_STATE and INSECURE_STATE are concerned with the state of the system when started or restarted, not the ongoing state of the system. The two threats also use nearly identical language. The term “unknown” is used in the former, found in the SLOS PP, while that term is changed to “insecure” in the separation kernel protection profile threat listing [SLP03, SKP04]. The latter favors the term “insecure state” over “unknown state” for the following reason. If the state of a system is unknown, this means it could be either secure or insecure. But if the system were in a secure state, and it could be determined that this state was secure, then the state is no longer unknown. Thus as long as the state is not known, it should be assumed that the state is insecure until proven otherwise. In other words, the idea is to err on the side of caution and choose to fail safe [LUN89]. This is why the term “insecure state” is favored in the separation kernel protection profile. One could argue that the threat UNKNOWN_STATE should have used this terminology as well. It is likely that this term was avoided since in a medium robustness system there is no notion of secure state. In contrast, a high robustness system would have a formal model proving that the system can only reach secure states. Due to their similarity, they will be considered equivalent for purposes of this recommendation.

a. Discussion

To demonstrate that the initialization routine does, in fact, reach an initial secure state, this recommendation proposes that first a precise definition or specification of the desired initial secure state be developed. This should include details such as the value in each of the general purpose registers, the values in segmentation registers, as well as the layout of memory. Then a mapping between the initialization implementation and this specification should be generated. Depending on the desired robustness level of the system this mapping may be more or less rigorous. For a medium robustness system, it may be a relatively informal document. For a high robustness system, a much more rigorous mapping on a line by line basis would be called for. This mapping combined with a carefully developed specification would serve to counter the similar threats UNKNOWN_STATE and INSECURE_STATE.

D. SUMMARY

At this point, the purpose of each recommendation, namely the threat that it counters, has been discussed. Each of these recommendations is based on the analysis presented in the proceeding chapters. While all of the recommendations should be used in development of a high assurance system, some would be too expensive or time-consuming to implement for a medium robustness system. Most recommendations were also accompanied by clear enforcement strategies, but some, such as enforcement of coding standards, can be difficult to do well. Despite the difficulty of realizing some of the recommendations presented here, they represent guidance to designers and developers of both current and future operating systems.

THIS PAGE INTENTIONALLY LEFT BLANK

VIII. CONCLUSIONS

A. FUTURE WORK

The scope of this thesis has necessarily been limited but gives rise to a number of topics for future work. As this thesis focuses solely on kernel space initialization tasks, additional research should explore the initialization tasks that are handled within user space, typically by the *init* binary. A question that should be asked is whether or not some of the tasks that are handled within user space could be more appropriately be handled within kernel space or vice versa.

The addition of hardware support to provide separation of initialization and runtime code could also be explored further. In recommendation R.1, it was mentioned that hardware support should ideally be used to ensure that initialization code only is executed a single time. The various privilege levels provided by the Intel chip would seem to provide this support. But in the current Intel architecture, only PL 0 is allowed to execute privileged instructions, instructions that both initialization and runtime code would need the ability to execute. Therefore implementing the initialization code in a different PL than the runtime code is currently not possible. To meet the above proposal, a new architecture would be required that provided two PLs that both had the ability to execute privileged instructions. The number of privileged instructions available within each PL could be minimized based on the nature of the code that would run within the given PL. For example, a new hardware instruction could be defined that allowed transfer of execution from the initialization segment to the runtime segment but no mechanism to go the other way. This would essentially provide an alternate way of achieving this recommendation.

A related idea that may be interesting to explore is the notion of a hardware co-processor whose sole purpose would be to verify that the system has reached an initial secure state. If such a co-processor could verify the state of the CPU and associated memory, then, in theory, an adversary could write the initialization routine without adverse effect. Detection of whether an initial secure state had been reached or not would be left to the co-processor. Yet such an architecture simply shifts the burden of verification and complexity. Assurance would still need to be provided that the co-processor functioned as required. But if the requirements of the co-processor could be sufficiently defined and implemented in hardware, this would provide a greater resistance to attacks on the integrity of the operating system kernel image, as a hardware solution would be more difficult for a remote attacker to defeat.

Another interesting and slightly related topic is that of malicious devices. To what extent does or should a system be able to protect itself against malicious logic found within the firmware of an attached device? Should the initialization routine provide protection against the presence of malicious devices or is this not realistic? While well beyond the scope of this thesis, these types of questions came up during the course of this research and may provide interesting directions for future work.

In R.2, it was mentioned that automated tools could provide important support for enforcement of coding standards. It would be useful to explore the tools currently available for this kind of purpose and determine the areas in which they may fall short. Future work could involve expanding the functionality of such tools.

While none of these proposed areas of additional work are small or simple, each reflects a different direction that could be taken based on the foundation work laid in this thesis.

B. SUMMARY

As the need for high assurance systems will only increase within the U.S. Department of Defense in the coming years, a through understanding of the requirements for building such systems is essential. The need for a wider understanding of how such systems can be built has been recognized by the Trusted Computing Exemplar project, [IRV04] at the Naval Postgraduate School. The goal of the project is to create an open worked example of a high-assurance kernel.

This thesis has contributed to an understanding of how to build such high assurance systems by shedding additional light on the initialization sequence and the role it plays in an operating system. Based on the analysis presented here, a number of recommendations for how to design and implement the initialization routine of an operating system have been drawn up. These recommendations are high-level, yet easily translate to real-world actions that can be taken in order to provide a greater degree of assurance within an operating system. The recommendations presented here should serve to inform developers about the particular issues surrounding the initialization sequence. As well, these recommendations highlight where additional future work should be focused in order to realize high assurance goals.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. TOOLS

A number of tools were used to aid the analysis presented in the preceding chapters. Common UNIX tools such as *grep* were useful for searching source code for key initialization markers, such as loading of the GDT register (*lgdt*). To aid in organizing the information gleaned from such searches, the notebook program NoteTaker, developed by AquaMinds, was used [NTK]. Using the AppleScript interface, a script was developed which takes input from the user about what string to search for and where to search, then runs the *grep* utility using those arguments and outputs the results into the current notebook. The content of the script is given below.

```
set theString to text returned of (display dialog
"Enter a string to search for:" default answer "")
set thePath to text returned of (display dialog
"Enter a path to search:" default answer "")
set r to "grep -nr " & theString & "
/Users/cdodge/sourceCode" & thePath
display dialog (do shell script r)
```

The primary piece of software used to view and navigate source code was SourceNavigator [SNAV]. This program produces a cross-referenced database of all functions and defined variables when a project comprised of many source code files is loaded. The program is then capable of quick navigation from where a function is called to the function's definition via a single keystroke. Similar capabilities apply to variables. SourceNavigator is available for both Windows and UNIX platforms. The UNIX version uses the X11 library to provide a graphical interface and was easily compiled for OS X v.10.3.6. The only configuration that needed to be done by hand was defining the TCL_LIBRARY environment variable correctly. For example, if using the TCL libraries provided by the fink project, this variable should be set to */sw/share/etc* [FINK]. Use of this product for any similar type of analysis would be highly recommended.

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

[ARB99] Arbaugh, W.A., “Chaining Layered Integrity Checks,” PhD Thesis, University of Pennsylvania, 1999.

[BOV02] Bovet, D. P. and Cesati, M., *Understanding the Linux Kernel*, 2nd edition, O'Reilly & Associates, 2002.

[CPQ94] Compaq Computer Corporation, Phoenix Technologies and Intel Corporation, *Plug and Play BIOS Specification*, Version 1.0A, May 1994.

[CPQ96] Compaq Computer Corporation, Phoenix Technologies and Intel Corporation, *BIOS Boot Specification*, Version 1.01, January 1996.

[DIJ68] Dijkstra, E.W., “The Structure of the “THE”-Multiprogramming System.” *ACM Symposium on Operating Systems Principles,* Vol. 11, No. 5, May 1968.

[FIA] Fiasco microkernel, http://os.inf.tu-dresden.de/fiasco/. Accessed October 2004.

[FINK] Fink project, http://fink.sourceforge.net/. Accessed December 2004.

[GCC04] Free Software Foundation, *GCC 3.4.1 Manual*, http://gcc.gnu.org/onlinedocs/gcc-3.4.1/gcc/. Accessed July 2004.

[GRUB] GRUB (Grand Unified Boot Loader), http://www.gnu.org/software/grub/. Accessed June 2004.

[GRU04] GRUB 2, http://www.gnu.org/software/grub/grub-2.en.html. Accessed November 2004.

[GRDV04] grub-devel mailing list, http://lists.gnu.org/archive/html/grub-devel/2004-05/msg00012.html. Accessed June 2004.

[IEEE90] IEEE Standard 610.12-1990. *IEEE Standard Glossary of Software Engineering Terminology*, IEEE Computer Society Press, 1990.

[INT03-2] Intel Corporation, *IA-32 Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, http://www.intel.com/design/pentium4/manuals/index_new.htm. Accessed May 2004.

[INT03-3] Intel Corporation, *IA-32 Intel Architecture Software Developer's Manual, Volume 2: System Programming Guide,* http://www.intel.com/design/pentium4/manuals/index_new.htm. Accessed May 2004.

[INT97] Intel Corporation, Nov. 20 1997. "Invalid Instruction Erratum Overview" Available from http://support.intel.com/support/processors/pentium/ppiie/. Accessed September 2004.

[IRV04] Irvine, C.E., Levin, T.E., Nguyen, T.D., and Dinolt, G.W., "The Trusted Computing Exemplar Project," *Proceedings of the 2004 IEEE Systems Man and Cybernetics Information Assurance Workshop*, West Point, NY, June 2004, pp. 109-115.

[LILO] LILO (The Linux Loader), http://freshmeat.net/projects/lilo/. Accessed May 2004.

[LUN89] Lunt, T.F., "Access Control Policies: Some Unanswered Questions," *Computers & Security*, No. 8, 1989, pp. 43-54.

[MACH] Mach operating system project, http://www-2.cs.cmu.edu/afs/cs/project/mach/public/www/mach.html. Accessed November 2004.

[MCK96] McKusick, M.K., Bostic, K., Karels, M.J., Quarterman, J.S.. *The Design and Implementation of the 4.4BSD Operating System*, Addison-Wesley, 1996.

[MIS01-1] OpenBSD misc mailing list, http://monkey.org/openbsd/archive/misc/0105/msg00684.html. Accessed November 2004.

[MIS01-2] OpenBSD misc mailing list, http://monkey.org/openbsd/archive/misc/0105/msg00725.html. Accessed November 2004.

[NTK] AquaMinds homepage for the NoteTaker product, http://www.aquaminds.com/product.jsp. Accessed December 2004.

[OBSD04] OpenBSD Security, http://www.openbsd.org/security.html. Accessed June 2004.

[OKU02] Okuji, Y.K, Ford, B., Boleyn, E.S., Ishiguro, K., *The Multiboot Specification*, Free Software Foundation, 2002. Available from http://www.gnu.org/software/grub/manual/multiboot/multiboot.pdf. Accessed May 2004.

[PAR72] Parnas, D.L., "On the Criteria To Be Used in Decomposing Systems into Modules," *Communications of the ACM*, v.15, n.12, pp. 1,053-1,058, December 1972.

[SCH96] Schneier, B., *Applied Cryptography*, 2nd edition, Wiley & Sons, New York, 1996.

[SKP04] NSA Information Assurance Directorate, *Protection Profile for Separation Kernels in Environments Requiring High Robustness*. July 1, 2004. Available from http://niap.nist.gov/pp/draft_pps/pp_draft_skpp_hr_v0.621.pdf. Accessed August 2004.

[SNAV] SourceNavigator project homepage, http://sourcenav.sourceforge.net/. Accessed December 2004.

[SLP03] NSA Information Assurance Directorate, *Protection Profile for Single-level Operating Systems in Environments Requiring Medium Robustness*. October 30, 2003. Available from http://niap.nist.gov/cc-scheme/pp/PP_SLOSPP-MR_V1.22.pdf. Accessed August 2004.

[TAN97] Tanenbaum, A.S., Woodhull, A.S., *Operating Systems: Design and Implementation*, 2nd edition, Prentice Hall, Upper Saddle River, New Jersey, 1997.

[TIS95] Tool Interface Standard (TIS) Committee, *Executable and Linking Format (ELF) Specification*, Version 1.2, http://www.x86.org/ftp/manuals/tools/elf.pdf. Accessed August 2004.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Hugo A. Badillo
   NSA
   Fort Meade, MD

4. George Bieber
   OSD
   Washington, DC

5. RADM Joseph Burns
   NSA
   Fort George Meade, MD

6. Deborah Cooper
   DC Associates, LLC
   Roslyn, VA

7. CDR Daniel L. Currie
   PMW 161
   San Diego, CA

8. LCDR James Downey
   NAVSEA
   Washington, DC

9. Dr. Diana Gant
   National Science Foundation
   Arlington, VA

10. Richard Hale
    DISA
    Falls Church, VA

11. LCDR Scott D. Heller
    SPAWAR
    San Diego, CA

12. Wiley Jones
OSD
Washington, DC

13. Russell Jones
N641
Arlington, VA

14. David Ladd
Microsoft Corporation
Redmond, WA

15. Dr. Carl Landwehr
National Science Foundation
Arlington, VA

16. Steve LaFountain
NSA
Fort Meade, MD

17. Dr. Greg Larson
IDA
Alexandria, VA

18. Penny Lehtola
NSA
Fort Meade, MD

19. Ernest Lucier
Federal Aviation Administration
Washington, DC

20. CAPT Sheila McCoy
Headquarters U.S. Navy
Arlington, VA

21. Dr. Vic Maconachy
NSA
Fort Meade, MD

22. Doug Maughan
Department of Homeland Security
Washington, DC

23. Dr. John Monastra
Aerospace Corporation
Chantilly, VA

24. John Mildner
SPAWAR
Charleston, SC

25. Keith Schwalm
Good Harbor Consulting, LLC
Washington, DC

26. Dr. Ralph Wachter
ONR
Arlington, VA

27. David Wirth
N641
Arlington, VA

28. Daniel Wolf
NSA
Fort Meade, MD

29. CAPT Robert Zellmann
CNO Staff N614
Arlington, VA

30. Dr. Cynthia E. Irvine
Naval Postgraduate School
Monterey, CA

31. Thuy Nguyen
Naval Postgraduate School
Monterey, CA

32. Catherine Dodge
Civilian, Naval Postgraduate School
Monterey, CA